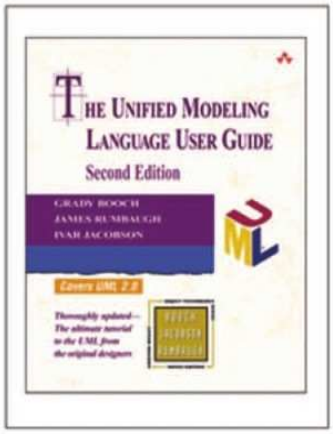
PEARSON

软件开发方法学
精选系列



Unified Modeling Language User Guide, Second Edition

[美] Grady Booch James Rumbaugh Ivar Jacobson 著
邵维忠 麻志毅 马浩海 刘辉 译

UML用户指南

（第2版·修订版）

人民邮电出版社
POSTS & TELECOM PRESS

目　录

封面
扉页
版权
版权声明
译者序
译者简介
前言
第一部分 入门
第1章 为什么要建模
1.1 建模的重要性
1.2 建模原理
1.3 面向对象建模
第2章 UML介绍
2.1 UML概述
2.1.1 UML是一种语言
2.1.2 UML是一种用于可视化的语言
2.1.3 UML是一种可用于详细描述的语言
2.1.4 UML是一种用于构造的语言
2.1.5 UML是一种用于文档化的语言
2.1.6 在何处能使用UML
2.2 UML的概念模型
2.2.1 UML的构造块
2.2.2 UML规则
2.2.3 UML中的公共机制
2.3 体系结构
2.4 软件开发生命周期

第3章 Hello,World！
3.1 关键抽象
3.2 机制
3.3 制品
第二部分 对基本结构建模
第4章 类
4.1 入门
4.2 术语和概念
4.2.1 名称
4.2.2 属性
4.2.3 操作
4.2.4 对属性和操作的组织
4.2.5 职责
4.2.6 其他特征
4.3 常用建模技术
4.3.1 对系统的词汇建模
4.3.2 对系统中的职责分布建模
4.3.3 对非软件事物建模
4.3.4 对简单类型建模
4.4 提示和技巧
第5章 关系
5.1 入门
5.2 术语和概念
5.2.1 依赖
5.2.2 泛化
5.2.3 关联
5.2.4 其他特征
5.2.5 绘图风格

5.3 常用建模技术
5.3.1 对简单依赖建模
5.3.2 对单继承建模
5.3.3 对结构关系建模
5.4 提示和技巧
第6章 公共机制
6.1 入门
6.2 术语和概念
6.2.1 注解
6.2.2 其他修饰
6.2.3 衍型
6.2.4 标记值
6.2.5 约束
6.2.6 标准元素
6.2.7 外廓
6.3 常用建模技术
6.3.1 对注释建模
6.3.2 对新特性建模
6.3.3 对新语义建模
6.4 提示和技巧
第7章 图
7.1 入门
7.2 术语和概念
7.2.1 结构图
7.2.2 行为图
7.3 常用建模技术
7.3.1 对系统的不同视图建模
7.3.2 对不同的抽象层次建模

7.3.3 对复杂视图建模
7.4 提示和技巧
第8章 类图
8.1 入门
8.2 术语和概念
8.2.1 普通特性
8.2.2 内容
8.2.3 一般用法
8.3 常用建模技术
8.3.1 对简单协作建模
8.3.2 对逻辑数据库模式建模
8.3.3 正向工程和逆向工程
8.4 提示和技巧
第三部分 对高级结构建模
第9章 高级类
9.1 入门
9.2 术语和概念
9.2.1 类目
9.2.2 可见性
9.2.3 实例范围和静态范围
9.2.4 抽象元素、叶子元素和多态性元素
9.2.5 多重性
9.2.6 属性
9.2.7 操作
9.2.8 模板类
9.2.9 标准元素
9.3 常用建模技术
9.4 提示和技巧

第10章 高级关系
10.1 入门
10.2 术语和概念
10.2.1 依赖
10.2.2 泛化
10.2.3 关联
10.2.4 实现
10.3 常用建模技术
10.4 提示和技巧
第11章 接口、类型和角色
11.1 入门
11.2 术语和概念
11.2.1 名称
11.2.2 操作
11.2.3 关系
11.2.4 理解接口
11.3 常用建模技术
11.3.1 对系统中的接缝建模
11.3.2 对静态类型和动态类型建模
11.4 提示和技巧
第12章 包
12.1 入门
12.2 术语和概念
12.2.1 名称
12.2.2 拥有的元素
12.2.3 可见性
12.2.4 引入与引出
12.3 常用建模技术

12.3.1 对成组的元素建模
12.3.2 对体系结构视图建模
12.4 提示和技巧
第13章 实例
13.1 入门
13.2 术语和概念
13.2.1 抽象和实例
13.2.2 类型
13.2.3 名称
13.2.4 操作
13.2.5 状态
13.2.6 其他特征
13.2.7 标准元素
13.3 常用建模技术
13.4 提示和技巧
第14章 对象图
14.1 入门
14.2 术语和概念
14.2.1 普通特性
14.2.2 内容
14.2.3 一般用法
14.3 常用建模技术
14.3.1 对对象结构建模
14.3.2 逆向工程
14.4 提示和技巧
第15章 构件
15.1 入门
15.2 术语和概念

15.2.1 构件和接口
15.2.2 可替换性
15.2.3 组织构件
15.2.4 端口
15.2.5 内部结构
15.3 常用建模技术
15.3.1 对结构类建模
15.3.2 对API建模
15.4 提示和技巧
第四部分 对基本行为建模
第16章 交互
16.1 入门
16.2 术语和概念
16.2.1 语境
16.2.2 对象和角色
16.2.3 链和连接件
16.2.4 消息
16.2.5 序列
16.2.6 创建、修改和撤销
16.2.7 表示法
16.3 常用建模技术
16.4 提示和技巧
第17章 用况
17.1 入门
17.2 术语和概念
17.2.1 主题
17.2.2 名称
17.2.3 用况与参与者

17.2.4 用况与事件流
17.2.5 用况与脚本
17.2.6 用况与协作
17.2.7 组织用况
17.2.8 其他特性
17.3 常用建模技术
17.4 提示和技巧
第18章 用况图
18.1 入门
18.2 术语和概念
18.2.1 公共特性
18.2.2 内容
18.2.3 表示法
18.2.4 一般用法
18.3 常用建模技术
18.3.1 对系统的语境建模
18.3.2 对系统的需求建模
18.3.3 正向工程和逆向工程
18.4 提示和技巧
第19章 交互图
19.1 入门
19.2 术语和概念
19.2.1 公共特性
19.2.2 内容
19.2.3 顺序图
19.2.4 顺序图中的结构化控制
19.2.5 嵌套活动图
19.2.6 通信图

19.2.7 语义等价
19.2.8 一般用法
19.3 常用建模技术
19.3.1 按时间顺序对控制流建模
19.3.2 按组织对控制流建模
19.3.3 正向工程和逆向工程
19.4 提示和技巧
第20章 活动图
20.1 入门
20.2 术语和概念
20.2.1 公共特性
20.2.2 内容
20.2.3 动作和活动结点
20.2.4 控制流
20.2.5 分支
20.2.6 分岔和汇合
20.2.7 泳道
20.2.8 对象流
20.2.9 扩展区域
20.2.10 一般用法
20.3 常用建模技术
20.3.1 对工作流建模
20.3.2 对操作建模
20.3.3 正向工程和逆向工程
20.4 提示和技巧
第五部分 对高级行为建模
第21章 事件和信号
21.1 入门

21.2 术语和概念
21.2.1 事件的种类
21.2.2 信号
21.2.3 调用事件
21.2.4 时间事件和变化事件
21.2.5 发送和接收事件
21.3 常用建模技术
21.3.1 对信号族建模
21.3.2 对异常建模
21.4 提示和技巧
第22章 状态机
22.1 入门
22.2 术语和概念
22.2.1 语境
22.2.2 状态
22.2.3 转移
22.2.4 高级状态和转移
22.2.5 子状态
22.3 常用建模技术
为对象的生命期建模
22.4 提示和技巧
第23章 进程和线程
23.1 入门
23.2 术语和概念
23.2.1 控制流
23.2.2 类和事件
23.2.3 通信
23.2.4 同步

23.3 常用建模技术
23.3.1 对多控制流建模
23.3.2 对进程间通信建模
23.4 提示和技巧
第24章 时间和空间
24.1 入门
24.2 术语和概念
24.2.1 时间
24.2.2 位置
24.3 常用建模技术
24.3.1 对定时约束建模
24.3.2 对对象的分布建模
24.4 提示和技巧
第25章 状态图
25.1 入门
25.2 术语和概念
25.2.1 公共特性
25.2.2 内容
25.2.3 一般用法
25.3 常用建模技术
25.3.1 对反应型对象建模
25.3.2 正向工程和逆向工程
25.4 提示和技巧
第六部分 对体系结构建模
第26章 制品
26.1 入门
26.2 术语和概念
26.2.1 名称

26.2.2 制品和类
26.2.3 制品的种类
26.2.4 标准元素
26.3 常用建模技术
26.3.1 对可执行程序和库建模
26.3.2 对表、文件和文档建模
26.3.3 对源代码建模
26.4 提示和技巧
第27章 部署
27.1 入门
27.2 概念和术语
27.2.1 名称
27.2.2 结点和制品
27.2.3 组织结点
27.2.4 连接
27.3 常用建模技术
27.3.1 对处理器和设备建模
27.3.2 对制品的分布建模
27.4 提示和技巧
第28章 协作
28.1 入门
28.2 术语和概念
28.2.1 名称
28.2.2 结构
28.2.3 行为
28.2.4 组织协作
28.3 常用建模技术
28.3.1 对角色建模

28.3.2 对用况的实现建模
28.3.3 对操作的实现建模
28.3.4 对机制建模
28.4 提示和技巧
第29章 模式和框架
29.1 入门
29.2 术语和概念
29.2.1 模式和体系结构
29.2.2 机制
29.2.3 框架
29.3 常用建模技术
29.3.1 对设计模式建模
29.3.2 对体系结构模式建模
29.4 提示和技巧
第30章 制品图
30.1 入门
30.2 术语和概念
30.2.1 普通特性
30.2.2 内容
30.2.3 一般用法
30.3 常用建模技术
30.3.1 对源代码建模
30.3.2 对可执行程序的发布建模
30.3.3 对物理数据库建模
30.3.4 对可适应系统建模
30.3.5 正向工程和逆向工程
30.4 提示和技巧
第31章 部署图

31.1 入门
31.2 术语和概念
31.2.1 普通特性
31.2 2 内容
31.2.3 一般用法
31.3 常用建模技术
31.3.1 对嵌入式系统建模
31.3.2 对客户/服务器系统建模
31.3.3 对全分布式系统建模
31.3.4 正向工程和逆向工程
31.4 提示和技巧
第32章 系统和模型
32.1 入门
32.2 术语和概念
32.2.1 系统和子系统
32.2.2 模型和视图
32.2.3 跟踪
32.3 常用建模技术
32.3.1 对系统的体系结构建模
32.3.2 对系统的系统建模
32.4 提示和技巧
第七部分 结束语
第33章 应用UML
33.1 转到UML
33.2 进一步介绍
附录A UML表示法
附录B Rational统一过程
术语表

索引
浅论科技术语翻译中的字面含义和技术含义

软件开发方法学精选系列

Unified Modeling Language User Guide,Second Edition

UML用户指南（第2版·修订版）

[美]Grady Booch　James Rumbaugh　Ivar Jacobson　著

邵维忠　麻志毅　马浩海　刘辉　译

人民邮电出版社

北京

图书在版编目（CIP）数据

UML用户指南：第2版/（美）布奇（BOOCH,.），（美）兰宝（Rumbaugh,.），（美）雅各布（Jacobson,I.）著；邵维忠等译.--北京：人民邮电出版社，2013.1

（软件开发方法学精选系列）

书名原文：Unified Modeling Language User Guide,Second Edition

ISBN　978-7-115-29644-3

Ⅰ.①U…　Ⅱ.①布…②兰…③雅…④邵…　Ⅲ.①面向对象语言—程序设计　Ⅳ.①TP312

中国版本图书馆CIP数据核字（2012）第242102号

内容提要

本书是UML方面的一部权威著作，3位作者是面向对象方法最早的倡导者、UML的创始人。本版涵盖了UML2.0。书中为UML具体特征的使用提供了指南，描述了使用UML进行开发的过程，旨在让读者掌握UML的术语、规则和惯用法，以及如何有效地使用这种语言，知道如何应用UML去解决一些常见的建模问题。本书由7个部分共33章组成，每章都对一组UML特征及其具体用法进行了详细阐述，其中大部分按入门、术语和概念、常用建模技术、提示和技巧的方式组织。本书还为高级开发人员提供了在高级建模问题中应用UML的一条非常实用的线索。

本书适合作为高等院校计算机及相关专业本科生或研究生“统一建模语言（UML）”课程的教材，也适合从事软件开发的工程技术人员和软件工程领域的研究人员参考。

软件开发方法学精选系列

UML用户指南（第2版·修订版）

◆著　[美]Grady Booch　James Rumbaugh　Ivar Jacobson

译　邵维忠　麻志毅　马浩海　刘辉

责任编辑　杨海玲
◆人民邮电出版社出版发行　　北京市崇文区夕照寺街14号
邮编　100061　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷
◆开本：800×1000　1/16
印张：24.25
字数：535千字　　2013年1月第1版
印数：1-3000册　　2013年1月北京第1次印刷
著作权合同登记号　图字：01-2012-7097号
ISBN　978-7-115-29644-3
定价：59.00元
读者服务热线：(010)67132692　印装质量热线：(010)67129223
反盗版热线：(010)67171154
广告经营许可证：京崇工商广字第0021号

版权声明

译者序

开发一个复杂的软件系统和编写一个简单的程序大不一样。其间的差别，借用G.Booch 的比喻，如同建造一座大厦和搭建一个狗窝的差别。大型的、复杂的软件系统开发是一项系统工程，必须按工程学的方法来组织软件生产，需要经过一系列的软件生命周期阶段。这是人们从软件危机中获得的最重要的教益。这一认识促使了软件工程学的诞生。编程仍然是重要的，但是更具有决定意义的是系统建模。只有在分析和设计阶段建立了良好的系统模型，才有可能保证工程的正确实施。由于这一原因，在编程领域出现的许多新方法和新技术，总是很快地拓展到软件生命周期的分析与设计阶段。

面向对象方法正是经历了这样的发展过程，它首先在编程领域兴起，作为一种崭新的程序设计范型引起世人瞩目。继Smalltalk-80之后，20世纪80年代有一大批面向对象编程语言问世，标志着面向对象方法走向成熟和实用。此时，面向对象方法开始向系统设计阶段延伸，出现了一批早期的面向对象设计（OOD）方法。到80年代末期，面向对象方法的研究重点转向面向对象的分析（OOA），并将 OOA 与 OOD 密切地联系在一起，出现了一大批面向对象的分析与设计（OOA&D）方法。至1994年，公开发表并具有一定影响的OOA&D方法已达50余种。这种繁荣的局面表明面向对象方法已经深入到分析与设计领域。此后，大多数比较成熟的软件开发组织已经从分析、设计到编程、测试全面地采用面向对象方法，使面向对象无可置疑地成为软件领域的主流技术。

各种OOA&D方法都为面向对象理论与技术的发展做出了贡献。这些方法的主导思想以及所采用的主要概念与原则大体上是一致的，但是也存在不少差异。这些差异所带来的问题是：不利于OO方法的发展，妨碍了技术交流，也给用户的选择带来困惑。在这种形势下，统一建模语言（Unified Modeling Language，UML）应运而生。

UML是在多种面向对象分析与设计方法相互融合的基础上形成的，其发展历史可以大致概括为4个阶段。最初的阶段是面向对象方法学家的联合行动，由 G.Booch、J.Rumbaugh和I.Jacobson将他们各自的方法结合起来，形成了UML0.9。第二阶段是公司的联合行动，由十多家公司组成 UML 伙伴组织，共同提出了 UML1.0 和 1.1，于 1997年被对象管理组织（OMG）正式采纳作为建模语言规范。第三阶段是在OMG控制下对UML规范进行修订和改进，产生了 UML1.2、1.3、1.4和 1.5等版本。第四阶段是从 1999年开始酝酿，并于本世纪初实施的一次重大的修订，推出了UML2.0，继而进行了多次修订，产生了UML2.1到UML2.4一系列版本，提交到国际标准化组织ISO作为建模语言标准的提案，其中各个部分已陆续进入ISO的标准化日程。

UML 用于对软件密集型系统进行详述、可视化、构造和文档化，也可以用于业务建模以及其他非软件系统的建模。UML 定义了系统建模所需的概念并给出其可视化表示法，但是它并不涉及如何进行系统建模。因此它只是一种建模语言，而不是一种建模方法。UML 是独立于过程的，就是说，它可以适应不同的建模过程。UML 的出现使面向对象建模概念和表示法趋于统一和标准化。目前UML已成为被广泛公认的工业标准，拥有越来越多的用户。现今大部分面向对象系统的建模均采用UML。

G.Booch、J.Rumbaugh和I.Jacobson是UML的3位主要奠基人，被称为“三友”，他们为UML 的形成和发展做出了卓越贡献。在广大读者的殷切期待中，“三友”联名撰写的 3 本介绍UML以及Rational统一软件

开发过程的著作（The Unified Modeling Language User Guide、The Unified Modeling Language Reference Manual和The Unified Software Development Process）于1999年由Addison-Wesley出版，深受广大读者的欢迎，被视为UML方面的权威性著作。在UML2.0问世之后，“三友”对他们的上述3本著作进行了再创作，以适应UML2.0的新内容，作为第2版，于2005年陆续出版。

现在我们翻译的《UML用户指南》第2版（The Unified Modeling Language User Guide,Second Edition）是“三友”上述3本著作中的一本，是阅读另外两本著作的基础。书中为如何使用UML提供了指南，旨在让读者掌握UML的术语、规则和惯用法，学会如何有效地使用UML进行开发，如何应用UML去解决常见的建模问题。实际上，这不仅仅是一部深入介绍UML的技术文献，而且处处闪烁着作者在方法学方面的真知灼见，凝结了作者在软件工程、面向对象方法、构件技术等诸多领域的经验和智慧。该书语言生动、深入浅出、实例丰富、图文并茂。对于想学习和使用 UML 的广大读者，这是一本难得的好书。该书的宗旨并不是全面地介绍 UML，也不是完整地介绍软件开发过程，这些内容属于“三友”的另外两本著作。

承担这样一本好书的翻译工作是1项愉快而又严肃的任务。尽管我们对UML进行过多年的研究，并且翻译过该书的第1版，但是在新版的翻译中仍不敢有驾轻就熟的心理。对翻译中遇到的一些疑难问题，往往要经过反复讨论，并通过对UML的进一步研究，才能获得比较准确的译法。忠实于原文是我们始终遵循的宗旨，但是原著中存在着个别前后不一致或者与UML规范不一致的现象，译文中采用了两种处理方式：对比较明显的错误在译文中做了订正，并通过译者注加以说明；对不太明显的错误按原文翻译，并在译者注中指出疑点。

本书的第一个译本于2006年6月由人民邮电出版社出版。承蒙广大读者的厚爱，先后9次印刷，累计印数达15000册。在此期间，UML2

的版本升级并未影响本书的适用性，因为本书的宗旨并不是全面地介绍UML某个版本的具体细节，而是引导读者学习和使用 UML。人民邮电出版社为满足广大读者的迫切需求，决定再次出版这本书，趁此机会我们对2006年的译稿进行了全面的审核和修订。修订范围涉及全书各章，以及前言、术语表、附录和译者注，对一些翻译不太准确或前后不一致的地方逐一做了订正，对文字上不够通顺的地方也进行了修改。

书中的科技术语译法以国标GB/T11457《信息技术 软件工程术语》 [1] 和我国计算机界权威性工具书《计算机科学技术百科全书》 [2] 为基准。其中有几个比较关键的术语（例如 use case 和classifier 等），一些曾经流行的不同译法使读者对这些术语的含义产生了截然不同的理解。关于这些术语的译法问题，我们在《中国计算机学会通讯》2010年第1期上刊登的一篇短文曾对此专门加以论述，我们也将这篇短文附在本书最后，供读者参考。

本书的翻译和相关研究得到了高可信软件技术教育部重点实验室和北京大学信息科学技术学院的大力支持。北京大学软件研究所建模研究小组所开展的研究工作对本书的翻译提供了可靠的依据。在此，谨向上述单位和相关个人致以衷心感谢。同时，我们诚恳地希望广大读者对书中可能存在的疏漏和错误之处给予批评和指正。

译者

2012年10月于北京

---

[1]. 国家技术监督局.信息技术 软件工程术语 GB/T 11457——2006.北京：中国标准出版社，2006

[2]. 张效祥主编，计算机科学技术百科全书（第二版）.北京：清华大学出版社，2005

译者简介

邵维忠 北京大学信息科学技术学院教授、博士生导师。1970年毕业于北京大学数学力学系，1979至1983年在计算机科学技术系任教并攻读硕士学位。早期主要从事操作系统和软件工程领域的研究。自1991年起注重面向对象建模方法的研究与教学，撰写和翻译了多部关于面向对象方法的学术著作。在软件工程环境、面向对象方法、建模语言、软件复用、构件技术和中间件技术等领域承担了多项国家高技术研究发展计划（863）项目、国家重大基础研究（973）项目和国家自然科学基金项目。曾获国家科技进步二等奖及多项国家部委级奖励。

麻志毅 博士，北京大学信息科学技术学院副教授，国家软件工程协会软件工程分会秘书长。主要研究领域为软件建模技术、模型驱动开发技术和软件工程支撑环境等。发表学术论文80余篇，出版著（译）作11部。曾获国家科技进步二等奖和国家科技攻关优秀成果奖等多项国家部委奖励。

马浩海 博士、教授、IBM资深软件工程师。2006年在北京大学信息学院获得计算机软件与理论专业理学博士学位。先后就职于内蒙古大学、Platform Computing和IBM Canada。已发表学术论文30余篇。主要研究领域为分布式计算和大规模数据处理、软件工程、面向对象技术、软件建模语言和模型驱动的软件开发技术。

刘辉 博士、副教授。2008年毕业于北京大学信息科学技术学院计算机软件与理论专业，获理学博士学位。同年进入北京理工大学计算机学院从事教学科研工作，入选校优秀青年教师资助计划。目前主要从事软件重构、软件演化、软件维护、软件测试等方面的研究和教学工作。现主持国家自然科学基金两项、教育部博士点基金一项、其他纵向科研项目4项。以第一作者在 IEEE Transactions on Software Engineering等期刊及ESEC\FSE、ASE等国际会议发表学术论文十余篇。个人主页：http://cs.bit.edu.cn/liuhui/，电子邮箱：Liuhui08@bit.edu.cn。

前言

统一建模语言（Unified Modeling Language，UML）是一种用于对软件密集型系统的制品进行可视化、详述、构造和文档化的图形语言。UML 给出了一种描绘系统蓝图的标准方法，其中既包括概念性的事物（如业务过程和系统功能），也包括具体的事物（如用特定的编程语言编写的类、数据库模式和可复用的软件构件）。

本书旨在教会读者如何有效地使用UML。

本书涵盖了UML 2.0 [1]。

目标

在本书中，读者将获益于以下几点。

明白UML是什么，不是什么，以及为什么UML对于开发软件密集型系统的过程非常重要。

掌握UML的术语、规则和惯用法，一般说来，还将学会如何有效地使用这种语言。

知道如何应用UML去解决许多常见的建模问题。

本书为UML具体特征的使用提供了参考资料，但它不是一本全面的UML参考手册。全面的参考请参阅我们编写的The Unified Modeling Language Reference Manual第2版（Rumbaugh、Jacobson、Booch合著，Addison-Wesley出版公司2005年出版）[2]。

本书描述了使用UML进行开发的过程，但并没有提供对于开发过程的完整参考资料。开发过程是 The Unified Software Development

Process（Jacobson、Booch、Rumbaugh 合著，Addison-Wesley出版公司1999年出版）[3] 一书的重点。

最后，本书提供了如何运用UML去解决许多常见建模问题的提示和技巧，但没有讲述如何去建模。本书类似于一本编程语言的用户指南，它教用户如何使用语言，而不教用户如何编程。

读者对象

进行软件开发、部署和维护的人员均可使用 UML。本书主要针对用 UML 进行建模的开发组成员，但它也适用于为了理解、建造、测试和发布一个软件密集型系统而一起工作的人员，他们要阅读这些模型。虽然这几乎包含了软件开发组织中的所有角色，但本书特别适合下述人员阅读：分析员和最终客户（他们要详细说明系统应该具有的结构和行为）、体系结构设计人员（他们设计满足上述需求的系统）、开发人员（他们把体系结构转换为可执行的代码）、质量保证人员（他们检验并确认系统的结构和行为）、库管理人员（他们创建构件并对构件进行编目）、项目及程序管理者（他们一般是把握方向的领导者，要进行有序的管理，并合理地分配资源，以保证系统的成功交付）。

使用本书的人员应该具有面向对象概念的基本知识。如果读者具有面向对象编程的经验或懂得面向对象的方法，就能更容易掌握本书内容，但这并不是必需的。

怎样使用本书

初次接触UML的开发人员最好按顺序阅读本书。第2章提出了UML的概念模型，读者应特别予以注意。所有的章节都是这样组织的——每一章建立在前面各章的内容之上，循序渐进。

至于正在寻求用UML解决常见建模问题的有经验的开发人员，可以按任意顺序阅读本书。读者应该特别注意在各章中提到的常见建模问题。

本书的组织及特点

本书主要由7个部分组成:

第一部分 入门

第二部分 对基本结构建模

第三部分 对高级结构建模

第四部分 对基本行为建模

第五部分 对高级行为建模

第六部分 对体系结构建模

第七部分 结束语

本书还包含两个附录：UML表示法概要和 Rational统一过程概要。在附录后，提供了一个常见术语表和一个索引。

每章都描述了针对UML具体特征的用法，其中的大部分按下述4节的方式组织:

（1）入门;

（2）术语和概念;

（3）常用建模技术;

（4）提示和技巧。

第3节“常用建模技术”提出一组常见建模问题并予以解决。为了便于读者浏览本书找到这些UML的应用场合，每一个问题都标有一个明显的标题，如下例所示。

对体系结构模式建模

每一章都从它所涵盖的特征概要开始，如下例所示。

本章内容

主动对象、进程和线程

对多控制流建模

对进程间通信建模
建立线程安全的抽象

类似地，把附加的解释和一般性的指导分离出来作为注解，如下例所示。

注解 UML 中的抽象操作对应于 C++中的纯虚操作；叶子操作对应于 C++的非虚操作。

UML的语义是非常丰富的，因此对一个特征的描述自然会涉及另一个特征。在这种情况下，在自然段的最后部分标注交叉引用，正如本段这样。

【第25章讨论构件。】

在图中使用灰色字 [4] 是为了表明这些文字不是模型本身的一部分，只是用于解释模型。程序代码用Courier字体表示以示区别，如this example。

致谢

作者向Bruce Douglass、Per Krol和Joaquin Miller表示感谢，谢谢他们帮助审阅了第2版的书稿。

UML简史

通常公认的第一种面向对象语言是1967年由Dahl和Nygaard在挪威开发的Simula-67。虽然该语言从来没有得到大量拥护者，但是它的概念给后来的语言以很大启发。Smalltalk在20世纪80年代早期得到了广泛的使用，到20世纪80年代晚期跟着出现了其他的面向对象语言，如Objective C、C++和Eiffel等。方法学家面对新型面向对象编程语言的涌现和不断增长的应用系统复杂性，开始试验用不同的方法来进行分析和设计，由此在20世纪80年代出现了面向对象建模语言。在1989年到1994年之间，面向对象的方法从不足10种增加到50多种。面对这么多的方法，很多用户很难找到一种完全满足他们要求的建模语言，于是就加剧了所谓的“方法战”。一些杰出的方法脱颖而出，其中包括

Booch方法、Jacobson的OOSE（面向对象的软件工程）和Rumbaugh 的OMT（对象建模技术）。其他的重要方法还有 Fusion 方法、Shlaer-Mellor 方法和Coad-Yourdon方法。这些方法中的每一种方法都是完整的，但是每一种方法又都被认为各有优点和缺点。简单来说，Booch方法在项目的设计和构造阶段的表达力特别强，OOSE对以用况作为一种途径来驱动需求获取、分析和高层设计提供了极好的支持，而OMT对于分析和数据密集型信息系统最为有用。

到20世纪90年代中期，一个关键的想法开始形成。当时Grady Booch（Rational软件公司）、James Rumbaugh（通用电气公司）、Ivar Jacobson（Objectory公司）和其他一些人开始从彼此的方法中取长补短，他们的共同成果开始在全球范围内被公认为是领导性的面向对象方法。作为Booch方法、OOSE方法和OMT方法的主要作者，促使我们 3个人创建统一建模语言的原因有3个。第一，我们的方法已经在朝着相互独立的方向演化，而我们希望它朝着一个方向演化，这样可以消除任何不必要的和不合理的潜在差别，因为这样的差别会加重用户的疑惑。第二，通过统一我们的方法，能够给面向对象的市场带来一定的稳定，能够让人们使用一种成熟的建模语言去设计项目，使工具开发人员把焦点集中于最有用的特征。第三，希望我们的合作能够改进早期的3种方法，帮助我们吸取教训，解决以前的方法不能妥善处理的问题。

统一工作之始，我们确立了3个工作目标。

（1）运用面向对象技术对系统进行从概念到可执行制品的建模。

（2）解决复杂系统和关键任务系统中固有的规模问题。

（3）创造一种人和机器都可以使用的建模语言。

设计一种用于面向对象分析和设计的语言与设计一种编程语言不同。首先，必须缩小问题范围：这种语言是否应包含需求描述？这种语言是否应支持可视化编程？其次，必须在表达能力和表达的简洁性

之间做好平衡。太简单的语言会限制能够解决问题的范围，而太复杂的语言会使开发人员无所适从。在统一现有方法的情况下，也必须小心从事。若对语言进行太多的改进，会给已有用户造成混乱；若不对语言进行改进，则会失去赢得更广大的用户群和使语言得到简化的时机。UML的定义力争在这些方面做出最好的选择。

1994年10月，Rumbaugh加入Booch所在的Rational公司，自此正式开始了UML的统一工作。我们的计划最初注重于联合Booch方法和OMT方法。“统一方法”（当时的名称）0.8版本（草案）在1995年10月发布。差不多就在那时，Jacobson也加入了Rational公司，于是UML项目的范围又做了扩充，把OOSE也结合进来。经过我们的努力，在1996年6月发布了UML 0.9版本。1996 年全年，我们都在软件工程界征求和收集反馈意见。在此期间，明显地有很多软件组织把UML作为商业战略来考虑。我们与几个愿意致力于定义一个强大而完善的UML的组织一起成立了一个UML伙伴组织。对UML 1.0版本做出贡献的合作伙伴有DEC、HP、I-Logix、Intellicorp、IBM、ICON Computing、MCI Systemhouse、Microsoft、Oracle、Rational、TI和Unisys。这些合作伙伴协作产生的UML 1.0版本是一个定义明确、富有表现力、强大、可应用于广泛问题域的建模语言。Mary Loomis帮助说服OMG（对象管理组织）发布了一个标准建模语言的提案需求（RFP）。在1997年1月，作为对该提案的响应，UML 1.0作为标准化的建模语言提交给OMG。

在1997年1月至7月之间，合作伙伴的队伍不断扩大，实际上包括了所有对最初OMG的提议做出贡献的公司，它们是 Andersen Consulting、Ericsson、ObjecTime Limited、Platinum Technology、PTech、Reich Technologies、Softeam、Sterling Software和Taskon。为了制定UML规范，并把 UML 与其他的标准化成果结合起来，成立了一支由 MCI Systemhouse 公司的 Cris Kobryn领导并由Rational公司的

Ed Eykholt管理的语义任务组。在1997年7月，把UML的修改版（1.1版本）提交给OMG，申请进行标准化审查。1997年9月，OMG的分析与设计任务组（Analysis and Design Task Force，ADTF）和OMG的体系结构部接受了该版本，并把它提交给OMG的全体成员进行表决。1997年11月14日，UML 1.1版本被OMG采纳。

几年来，UML一直由OMG的修订任务组（Revision Task Force，RTF）维护，陆续研发了UML的1.3、1.4和1.5版本。从2000年到2003年，一个经过扩充了的新的伙伴组织制定了一个升级的UML规范，即UML 2.0。由IBM的Bran Selic领导的定案任务组（Finalization Task Force，FTF）对这个版本进行了为期一年的评审，UML 2.0的正式版本于2005年初被OMG采纳。UML 2.0是对UML 1的重大修订，包括了大量的新增特性。此外，基于先前版本的经验，UML 2.0对先前版本的构造物做了很多的修改。可以在OMG的网站www.omg.org上获得当前的UML规范文档。

UML 是很多人的工作成果，它的思想来自于大量的先前工作。重新构造一个贡献者的完整列表将是一项很大的历史性研究工程，根据对UML影响大小来识别那么多的先驱者就更为困难了。同所有的科学研究和工程实践一样，UML只是站在巨人肩上而已。

---

[1]. UML2.0发布之后版本已多次更新，各个版本统称为UML2，目前最新的版本是UML2.4。本书绝大部分内容对UML2.0之后的各个版本仍然适应。——译者注

[2]. 中文版已由机械工业出版社出版，中文书名《UML参考手册》。——编者注

[3]. 中文版已由机械工业出版社出版，中文书名《统一软件开发过程》。——编者注

[4]. 本书中用灰色字代替原版中的蓝字表示这种区别。——编者注

第一部分 入门

第1章 为什么要建模

本章内容

建模的重要性

建模的4项原理

软件系统的基本蓝图

面向对象建模

成功的软件组织应该总是能够交付满足其用户需要的软件。如果一个软件组织能够及时并可预测地开发出这样的软件，并能够有效地利用人力和物力资源，那么这个软件组织就是可持续发展的。

在上段话里有一个重要的含义：一个开发队伍的主要产品不应该是一堆漂亮的文档、世界级的会议、伟大的口号或者几行获得普利策奖金的源代码，而应该是满足不断发展的用户及其业务需要的优秀软件。其他的一切事情都是次要的。

不幸的是，很多软件组织把“次要的”和“不重要的”的含义搞混了。为了得到满足预期功能的软件，必须到用户中去，以一种训练有素的方式访问用户，去揭示系统的真实需求。为了开发出具有持久质量的软件，必须打好能适应变化的、坚实的体系结构基础。为了能快速、有效地开发软件，尽量减少软件废品和重复工作，必须有合适的人员和合适的工具以及合适的工作重点。为了能一贯地、可预测地做到这

些，并使得在整个系统的生命期内花费合理，必须有一个能适应业务和技术变化的合理的开发过程。

建模是开发优秀软件的所有活动中的核心部分，其目的是为了把想要得到的系统结构和行为沟通起来，为了对系统的体系结构进行可视化和控制，为了更好地理解正在构造的系统，并经常揭示简化和复用的机会，同时也是为了管理风险。

1.1 建模的重要性

如果想搭一个狗窝，备好木料、钉子和一些基本工具（如锤子、锯和卷尺）之后，就可以开始工作了。从制订一点初步计划到完成一个满足适当功能的狗窝，可能不用别人帮助，在几个小时内就能够实现。只要狗窝够大且不太漏水，狗就可以安居。如果未能达到希望的效果，返工总是可以的，无非是让狗受点委屈。

如果想为家庭建造一所房子，备好木料、钉子和一些基本工具之后，也能开始工作，但这将需要较长的时间，并且家庭对于房子的需求肯定比狗对于狗窝的需求要多。在这种情况下，除非曾经多次建造过房子，否则就需要事先制定出一些详细的计划，再开始动工，才能够成功。至少应该绘制一些表明房子是什么样子的简图。如果想建造一所能满足家庭的需要并符合当地建筑规范的合格房屋，就需要画一些建筑图，以便能想清楚房间的使用目的以及照明、取暖和水管装置的实际细节问题。做出这些计划后，就能对这项工作所需的时间和物料做出合理的估计。尽管自己也可能建造出这样的房屋，但若有其他人协作，并将工程中的许多关键部分转包出去或购买预制的材料，效率就会高得多。只要按计划行事，不超出时间和财务的预算，家庭多半会对这新房感到满意。如果不制定计划，新房就不会完全令人满意。因此，最好在早期就制定计划，并谨慎地处理好所发生的变化。

如果你要建造一座高层办公大厦，若还是先备好木料、钉子和一些基本工具就开始工作，那将是非常愚蠢的。因为你所使用的资金可能是别人的，他们会对建筑物的规模、形状和风格做出要求。同时，他们经常会改变想法，甚至是在工程已经开工之后。由于失败的代价太高了，因此必须要做详尽的计划。负责建筑物设计和施工的是一个庞大的组织机构，你只是其中的一部分。这个组织将需要各种各样的设计图和模型，以供各方相互沟通。只要得到了合适的人员和工具，并对把建筑概念转换为实际建筑的过程进行积极的管理，将会建成这座满足使用要求的大厦。如果想继续从事建筑工作，那么一定要在使用要求和实际的建筑技术之间做好平衡，并且处理好建筑团队成员们的休息问题，既不能把他们置于风险之中，也不能驱使他们过分辛苦地工作以至于精疲力尽。

奇怪的是，很多软件开发组织开始想建造一座大厦式的软件，而在动手处理时却好像他们正在仓促地造一个狗窝。

有时你是幸运的。如果在恰当的时间有足够的合适人员，并且其他一切事情都很如意，你的团队有可能（仅是可能）推出一个令用户眼花缭乱的软件产品。然而，一般的情况下，不可能所有人员都合适（合适的人员经常供不应求），时间并不总是恰当的（昨天总是更好），其他的事情也并不尽如人意（常常由不得自己）。现在对软件开发的要求正在日益增加，而开发团队却还是经常单纯地依靠他们唯一真正知道如何做好的一件事——编写程序代码。英雄式的编程工作成为这一行业的传奇，人们似乎经常认为更努力地工作是面对开发中出现的各种危机的正常反应。然而，这未必能产生正确的程序代码，而且一些项目是非常巨大的，无论怎样延长工作时间，也不足以完成所需的工作。

如果真正想建造一个相当于房子或大厦类的软件系统，问题可不是仅仅编写许多软件。事实上，关键是要编出正确的软件，并考虑如

何少写软件。要生产合格的软件就要有一套关于体系结构、过程和工具的规范。即使如此，很多项目开始看起来像狗窝，但随后发展得像大厦，原因很简单，它们是自己成就的牺牲品。如果对体系结构、过程或工具的规范没有作任何考虑，总有一天狗窝会膨胀成大厦，并会由于其自身的重量而倒塌。狗窝的倒塌可能使你的狗恼怒；同理，不成功的大厦则将对大厦的租户造成严重的影响。

不成功的软件项目失败的原因各不相同，而所有成功的项目在很多方面都是相似的。成功的软件组织有很多成功的因素，其中共同的一点就是对建模的采用。

建模是一项经过检验并被广为接受的工程技术。建立房屋和大厦的建筑模型，能帮助用户得到实际建筑物的印象，甚至可以建立数学模型来分析大风或地震对建筑物造成的影响。

建模不只适用于建筑业。如果不首先构造模型（从计算机模型到物理风洞模型，再到与实物大小一样的原型），就装配新型的飞机或汽车，那简直是难以想象的。新型的电气设备（从微处理器到电话交换系统）需要一定程度的建模，以便更好地理解系统并与他人交流思想。在电影业，情节串联板是产品的核心，这也是建模的一种形式。在社会学、经济学和商业管理领域也需要建模，以证实人们的理论或用最小限度的风险和代价试验新的理论。

那么，模型是什么？简单地说：

模型是对现实的简化。

模型提供了系统的蓝图。模型既可以包括详细的计划，也可以包括从很高的层次考虑系统的总体计划。一个好的模型包括那些有广泛影响的主要元素，而忽略那些与给定的抽象水平不相关的次要元素。每个系统都可以从不同的方面用不同的模型来描述，因而每个模型都是一个在语义上闭合的系统抽象。模型可以是结构性的，强调系统的组织。它也可以是行为性的，强调系统的动态方面。

为什么要建模？一个基本理由是：

建模是为了能够更好地理解正在开发的系统。

通过建模，要达到以下4个目的。

（1）模型有助于按照实际情况或按照所需要的样式对系统进行可视化。

（2）模型能够规约系统的结构或行为。

（3）模型给出了指导构造系统的模板。

（4）模型对做出的决策进行文档化。

【第2章讨论UML如何完成这4件事情。】

建模并不只是针对大的系统。甚至像狗窝那样的软件也能从一些建模中受益。然而，可以明确地讲，系统越大、越复杂，建模的重要性就越大，一个很简单的原因是：

因为不能完整地理解一个复杂的系统，所以要对它建模。

人对复杂问题的理解能力是有限的。通过建模，缩小所研究问题的范围，一次只着重研究它的一个方面，这就是Edsger Dijkstra几年前讲的“分而治之”的基本方法，即把一个困难问题划分成一系列能够解决的小问题；解决了这些小问题也就解决了这个难题。此外，通过建模可以增强人的智力。一个适当选择的模型可以使建模人员在较高的抽象层次上工作。

任何情况下都应该建模的说法并没有落到实处。事实上，一些研究指出，大多数软件组织没有做正规的建模，即使做了也很少。按项目的复杂性划分一下建模的使用情况，将会发现：项目越简单，采用正规建模的就越少。

这里强调的是“正规”这个词。实际上，开发者甚至对非常简单的项目也要做一些建模工作，虽然很不正规。开发者可能在一块黑板上或一小片纸上勾画出他的想法，以对部分系统进行可视化表示，或者开发组可能使用CRC卡片描述一个场景或某种机制的设计。使用任何

一种这样的模型都没有什么错。如果它能行得通，就可以使用。然而，这些非正规的模型经常是太随意了，它没有提供一种容易让他人理解的共同语言。建筑业、电机工程业和数学建模都有通用的建模语言，在软件开发中使用一种共同的建模语言进行软件建模同样能使开发组织获益匪浅。

每个项目都能从一些建模中受益。即使在一次性的软件开发中——由于可视化编程语言的支持，可以轻而易举地扔掉不适合的软件。建模也能帮助开发组更好地对系统计划进行可视化，并帮助他们正确地进行构造，使开发工作进展得更快。如果根本不去建模，项目越复杂，就越有可能失败或者构造出错误的东西。所有实用系统都有一个自然趋势：随着时间的推移变得越来越复杂。虽然今天可能认为不需要建模，但随着系统的演化，终将会对这个决定感到后悔，但那时为时已晚。

1.2 建模原理

各种工程学科都有其丰富的建模运用历史。这些经验形成了建模的四项基本原理，现分别叙述如下。

第一，选择要创建什么模型，对如何动手解决问题和如何形成解决方案有着意义深远的影响 。

换句话说，就是要好好地选择模型。正确的模型将清楚地表明最棘手的开发问题，提供不能轻易地从别处获得的洞察力；错误的模型将使人误入歧途，把精力花在不相关的问题上。

暂时先把软件问题放在一边，假设现在正试图解决量子物理学上的一个问题。诸如光子在时空中的相互作用问题，其中充满了令人惊奇的难解的数学问题。选择一个不同的模型，所有的复杂问题一下子就变得可行了（虽然不容易解决）。在这个领域中，这恰恰是费曼图

的价值，它提供了对非常复杂问题的图形表示。类似地，在一个完全不同的领域里，假设正在建造一座新建筑，将会关心疾风对它的影响。如果建立了一个物理模型，并拿到风洞中去实验，虽然小模型没有精确地反映出大的实物，但也可以从中找出一些有趣的东西。因此，如果正在建立一个数学模型，然后去模拟，将知道一些不同的东西；与使用物理模型相比，也可能获得更多新的场景。通过对模型进行严格的持续的实验，将更信任已经建模的系统，事实上，它在现实世界中将像期望的那样工作得很好。

对于软件而言，所选择的模型将在很大程度上影响对领域的看法。如果以数据库开发者的观点建造一个系统，可能会注意实体——联系模型，该模型把行为放入触发器和存储过程中。如果以结构化开发者的观点建造一个系统，可能得到以算法为中心的模型，其中包含从处理到处理的数据流。如果以面向对象开发者的观点建造一个系统，将可能得到这样一个系统：它的体系结构以一组类和交互模式（指出这些类如何一起工作）为中心。可执行的模型对测试有很大帮助。上述的任何一种方法对于给定的应用系统和开发文化都可能是正确的，然而经验表明，在构建有弹性的体系结构中面向对象的方法表现得更为出众，即使对使用大型数据库或计算单元的系统也是如此。尽管如此，但要强调一点，不同的方法将导致不同种类的系统，并且代价和收益也是不同的。

第二，可以在不同的精度级别上表示每一种模型 。

如果正在建造一座大厦，有时需要从宏观上让投资者看到大厦的样子，感觉到大厦的总体效果。而有时又需要认真考虑细节问题，例如，对复杂棘手的管道的铺设，或对罕见的结构件的安装等。

对于软件模型也是如此。有时一个快速简洁且是可执行的用户界面模型正是所需要的，而有时必须耐着性子对付比特，例如，描述跨系统接口或解决网络瓶颈问题就是如此。在任何情况下，最好的模型

应该是这样的：它可以让你根据谁在进行观察以及为什么要观察选择它的详细程度。分析人员或最终用户主要考虑“做什么”的问题，开发人员主要考虑“怎样做”的问题。这些人员都要在不同的时间以不同的详细程度对系统进行可视化。

第三，最好的模型是与现实相联系的 。

如果一个建筑物的物理模型不能反映真实的建筑物，则它的价值是很有限的；飞机的数学模型，如果只是假定了理想条件和完美制造，则可能掩盖真实飞机的一些潜在的、致命的现实特征。最好是拥有能够清晰地联系实际的模型，而当联系很薄弱时能够精确地知道这些模型如何与现实脱节。所有的模型都对现实进行了简化；诀窍是，确保这种简化不要掩盖掉任何重要的细节。

软件领域中结构化分析的致命弱点是在分析模型和系统设计模型之间没有基本的联系。随着时间的推移，这个不可填充的裂缝会使系统构思阶段和实施阶段出现不一致。在面向对象的系统中，可以把各个几乎独立的系统视图连结成一个完整的语义整体。

第四，单个模型或视图是不充分的。对每个重要的系统最好用一小组几乎独立的模型从多个视角去逼近 。

如果正在建造一所建筑物，会发现没有任何一套单项设计图能够描述该建筑的所有细节。至少需要楼层平面图、立面图、电气设计图、采暖设计图和管道设计图。并且，在任何种类的模型中都需要从多视角来把握系统的范围（例如不同楼层的蓝图）。

在这里的重要短语是“几乎独立的”。在这个语境中，它意味着各种模型能够被分别进行研究和构造，但它们仍然是相互联系的。如同建造建筑物一样，既能够单独地研究电气设计图，但也能看到它如何映射到楼层平面图中，以及它与管道设计图中的管子排布的相互影响。

面向对象的软件系统也如此。为了理解系统的体系结构，需要几个互补和连锁的视图：用况视图（揭示系统的需求）、设计视图（捕获问题空间和解空间里的词汇）、交互视图 [1]（展示系统各部分之间以及系统与环境之间的联系）、实现视图（描述系统的物理实现）和部署视图（着眼于系统的工程问题）。每一种视图都可能有结构方面和行为方面。这些视图一起从整体上描绘了软件蓝图。

【第2章讨论这5种视图。】

根据系统的性质，一些模型可能比另一些模型要重要。例如，对于数据密集型系统，表达静态设计视图的模型将占主导地位；对于图形用户界面密集型系统，静态和动态的用况视图就显得相当重要；在硬实时系统中，动态进程视图尤为重要；最后，在分布式系统中，例如 Web 密集型的应用，实现模型和部署模型是最重要的。

1.3 面向对象建模

土木工程师构造了很多种模型。通常这些模型能帮助人们可视化并说明系统的各部分以及这些部分之间的相互关系。根据业务或工程中所着重关心的内容（例如为了帮助研究一个结构在地震时的反应）工程师也可以建立动态模型。各种模型的组织是不同的，各有自己的侧重点。对于软件，有好几种建模的方法。最普通的两种方法是从算法的角度建模和从面向对象的角度建模。

传统的软件开发是从算法的角度进行建模。按照这种方法，所有的软件都用过程或函数作为其主要构造块。这种观点导致开发人员把精力集中于控制流程和对大的算法进行分解。这种观点除了常常产生脆弱的系统之外没有其他本质上的害处。当需求发生变化（总会变化的）以及系统增长（总会增长的）时，用这种方法建造的系统就会变得很难维护。

现代的软件开发采用面向对象的观点进行建模。按照这种方法，所有软件系统都用对象或类作为其主要构造块。简单地讲，对象通常是从问题空间或解空间的词汇中抽取出来的东西；类是对具有共同性质的一组对象（从建模者的视角）的描述。每一个对象都有标识（能够对它命名，以区别于其他对象）、状态（通常有一些数据与它相联系）和行为（能对该对象做某些事，它也能为其他对象做某些事）。

例如，可考虑把一个简单的计账系统的体系结构分成3层：用户界面层、业务服务层和数据库层。在用户界面层，将找出一些具体的对象，如按钮、菜单和对话框。在数据库层，将找出一些具体的对象，例如描述来自问题域实体的表，包括顾客、产品和订单等。在中间层，将找出诸如交易、业务规则等对象，以及顾客、产品和订单等问题实体的高层视图。

可以肯定地说，面向对象方法是软件开发方法的主流部分，其原因很简单，因为事实已经证明，它适合于在各种问题域中建造各种规模和复杂度的系统。此外，当前的大多数程序语言、操作系统和工具在一定程度上都是面向对象的，并给出更多按对象来观察世界的理由。面向对象的开发为使用构件技术（如J2EE或.NET）装配系统提供了概念基础。

选择以面向对象的方式观察世界，会产生一系列的问题：什么是好的面向对象的体系结构？项目会创造出什么样的制品？谁创造它们？怎样度量它们？

【第2章讨论这些问题。】

对面向对象系统进行可视化、详述、构造和文档化正是统一建模语言（UML）的目的。

第2章 UML介绍

本章内容

UML概述

理解UML的3个步骤

软件体系结构

软件开发过程

统一建模语言（Unified Modeling Language，UML）是一种绘制软件蓝图的标准语言。可以用UML对软件密集型系统的制品进行可视化、详述、构造和文档化。

从企业信息系统到基于Web的分布式应用，乃至硬实时嵌入式系统，都适合用UML来建模。UML是一种富有表达力的语言，可以描述开发所需要的各种视图，然后以此为基础来部署系统。虽然 UML 的表达力很丰富，但理解和使用它并不困难。要学习使用 UML，一个有效的出发点是形成该语言的概念模型，这要求学习3个要素：UML的基本构造块、支配这些构造块如何放置在一起的规则以及运用于整个语言的一些公共机制。

UML仅仅是一种语言，因此仅仅是软件开发方法的一部分。UML是独立于过程的，但最好把它用于以用况为驱动、以体系结构为中心、迭代和增量的过程。

2.1 UML概述

UML是一种对软件密集型系统的制品进行下述工作的语言：

可视化；

详述；

构造；

文档化。

2.1.1 UML是一种语言

语言提供了用于交流的词汇表和在词汇表中组合词汇的规则，而建模语言的词汇表和规则注重于对系统进行概念上和物理上的描述，因而像UML这样的建模语言是用于软件蓝图的标准语言。

建模是为了产生对系统的理解。只用一个模型是不够的，相反，为了理解系统（除非是非常微小的系统）中的各种事物，经常需要多个相互联系的模型。对于软件密集型系统，就需要这样一种语言，它贯穿于软件开发的生命期，表达系统体系结构的各种不同视图。

【第1章讨论建模的基本原理。】

像UML这样的语言的词汇表和规则可以告诉你如何创建或理解形式良好的模型，但它没有说明应该在什么时候创建什么样的模型，因为这是软件开发过程的工作。一个定义良好的过程将指导你决定生产什么制品，由什么样的活动和人员来创建与管理这些制品，怎样采用这些制品从整体上去度量和控制项目。

2.1.2 UML是一种用于可视化的语言

对于很多程序员来说，从考虑实现到产生程序代码，其间没有什么距离可言，就是思考和编码。事实上，对有些事情的处理最好就是直接编码。使用文本是既省事又直接的书写表达式和算法的方式。

在这种情况下，程序员仍然要做一些建模，虽然只是在内心里这样做。他们甚至可以在白板或餐巾纸上草拟出一些想法。然而，这样做存在几个问题。第一，别人对这些概念模型容易产生错误的理解，因为并不是每个人都使用相同的语言。一种典型的情况是，假设项目开发单位建立了自己的语言，如果你是外来者或是加入项目组的新人，就难以理解该单位在做什么事。第二，除非建立了模型（不仅仅是文字的编程语言），否则就不能够理解软件系统中的某些事情。例如，阅读一个类层次的所有代码，虽可推断出它的含义，但不能直接领会它。类似地，在基于 Web的系统中研究系统的代码，虽可推断出

对象的物理分布和可能迁移，但也不能直接领会它。第三，如果一个开发者删节了代码而没有写下他头脑中的模型，一旦他另谋高就，那么这些信息就会永远丢失，最好的情况也只能是通过实现而部分地重建。

用UML建模可解决第三个问题：清晰的模型有利于交流。

对有些事物最好是用文字建模，而对有些事物又最好是用图形建模。的确，在所有引人关注的系统中都有一些用编程语言难以描绘的结构。UML 正是这样的图形化语言。这一点针对前面谈到的第二个问题。

UML不仅只是一组图形符号。确切地讲，UML表示法中的每个符号都有明确语义。这样，一个开发者可以用UML绘制一个模型，而另一个开发者（甚至工具）可以无歧义地解释这个模型。这一点针对前面谈到的第一个问题。

【UML的完整语义在The Unified Modeling Language Reference一书中讨论。】

2.1.3 UML是一种可用于详细描述的语言

在此处，详细描述意味着所建的模型是精确的、无歧义的和完整的。特别是，UML 适于对所有重要的分析、设计和实现决策进行详细描述，这些是软件密集型系统在开发和部署时所必需的。

2.1.4 UML是一种用于构造的语言

UML 不是一种可视化的编程语言，但用 UML 描述的模型可与各种编程语言直接相关联。这意味着一种可能性，即可把用UML描述的模型映射成编程语言，如Java、C++和Visual Basic等，甚至映射成关系数据库的表或面向对象数据库的持久存储。对一个事物，如果表示为图形方式最为恰当，则用UML，而如果表示为文字方式最为恰当，则用编程语言。

这种映射允许进行正向工程——从 UML 模型到编程语言的代码生成，也可以进行逆向工程——由编程语言代码重新构造UML模型。逆向工程并不是魔术。除非对实现中的信息编码，否则从模型到代码生成将会丢失信息。逆向工程需要工具支持和人的干预。把正向代码生成和逆向工程这两种方式结合起来就可以产生双向工程，这意味着既能在图形视图下工作，又能在文字视图下工作，只要用工具来保持二者的一致性即可。

【本书的第二部分和第三部分讨论对系统的结构建模。】

除了直接映射以外，UML 具有丰富的表达力，而且无歧义性，这允许直接执行模型、模拟系统以及对运行系统进行操纵。

【本书的第四部分和第五部分讨论对系统的行为建模。】

2.1.5 UML是一种用于文档化的语言

一个健康的软件组织除了生产可执行代码之外，还要给出各种制品。这些制品包括（但不限于）：

需求；

体系结构；

设计；

源代码；

项目计划；

测试；

原型；

发布。

依赖于开发文化，一些制品做得或多或少地比另一些制品要正规些。这些制品不但是项目交付时所要求的，而且无论是在开发期间还是在交付使用后对控制、度量和理解系统也是关键的。

UML适于建立系统体系结构及其所有细节的文档。UML还提供了用于表达需求和用于测试的语言。此外，UML提供了对项目计划活动和发布管理活动进行建模的语言。

2.1.6 在何处能使用UML

UML主要用于软件密集型系统。在下列领域中已经有效地应用了UML：

企业信息系统；

银行与金融服务；

电信；

运输；

国防/航天；

零售；

医疗电子；

科学；

基于Web的分布式服务。

UML 不限于对软件建模。事实上，它的表达能力对非软件系统建模也是足够的。例如，法律系统的工作流程、病人保健系统的结构和行为、飞机战斗系统中的软件工程以及硬件设计等。

2.2 UML的概念模型

为了理解UML，需要形成该语言的概念模型，这要求学习建模的3个要素：UML的基本构造块、支配这些构造块如何放在一起的规则和一些运用于整个UML的公共机制。如果掌握了这些思想，就能够读懂UML模型，并能建立一些基本模型。当有了较丰富的应用UML的经验时，就能够在这些概念模型之上使用更高深的语言特征进行构造。

2.2.1 UML的构造块

UML的词汇表包含下面3种构造块：

（1）事物；

（2）关系；

（3）图。

事物是对模型中首要成分的抽象，关系把事物结合在一起，图聚集了相关的事物。

1.UML中的事物

在UML中有4种事物：

（1）结构事物；

（2）行为事物；

（3）分组事物；

（4）注释事物。

这些事物是UML中基本的面向对象的构造块，用它们可以写出形式良好的模型。

2.结构事物

结构事物 （structural thing）是UML模型中的名词。它们通常是模型的静态部分，描述概念元素或物理元素。结构事物总称为类目（classifier）。

第一，类 （class）是对一组具有相同属性、相同操作、相同关系和相同语义的对象的描述。类实现一个或多个接口。在图形上，把类画成一个矩形，矩形中通常包括类的名称、属性和操作，如图2-1所示。

【第4章和第9章讨论类。】

第二，接口 （interface）是一组操作的集合，其中的每个操作描述了类或构件的一个服务。因此，接口描述了元素的外部可见行为。一个接口可以描述一个类或构件的全部行为或部分行为。接口定义了一组操作规约（即操作的特征标记），而不是操作的实现。接口的声明

看上去像一个类，在名称的上方标注着关键字«interface»；除非有时用来表示常量，否则不需要属性。然而，接口很少单独出现。如图2-2 所示，把由类提供的对外接口表示成用线连接到类框的一个小圆圈，把类向其他类请求的接口表示成用线连接到类框的半个小圆圈。

【第11章讨论接口。】

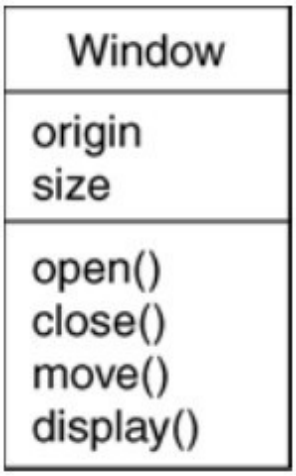


图2-1 类

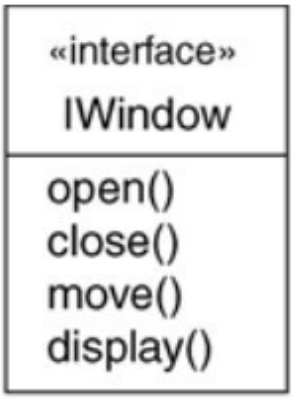


图2-2 接口

第三，协作（collaboration）定义了一个交互，它是由一组共同工作以提供某种协作行为的角色和其他元素构成的一个群体，这些协作行为大于所有元素的各自行为的总和。协作具有结构、行为和维度。一个给定的类或对象可以参与几个协作。这些协作因而表现了系统构成模式的实现。在图形上，把协作画成虚线椭圆，有时仅包含它的名称，如图2-3所示。

【第28章讨论协作。】

第四，用况（use case）是对一组动作序列的描述，系统执行这些动作将产生对特定的参与者有价值而且可观察的结果。用况用于构造模型中的行为事物。用况是通过协作实现的。在图形上，把用况画成实线椭圆，通常仅包含它的名称，如图2-4所示。

【第17章讨论用况。】

图2-3 协作

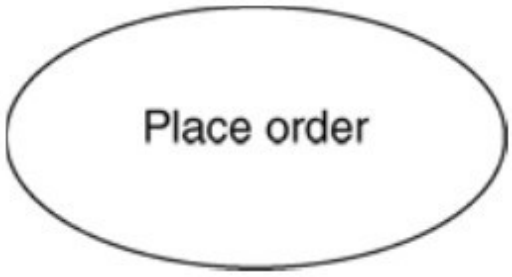


图2-4 用况

剩余的3种事物——主动类、构件和结点，都和类相似，就是说它们也描述了一组具有相同属性、操作、关系和语义的实体。然而，这3

种事物与类的不同点也不少，而且对面向对象系统的某些方面的建模是必要的，因此对这几个术语需要单独处理。

第五，主动类 （active class）是这样的类，其对象至少拥有一个进程或线程，因此它能够启动控制活动。主动类的对象所表现的元素的行为与其他元素的行为并发，除了这一点之外，它和类是一样的。在图形上，把主动类绘制成类图符，只是它的左右外框是双线，通常它包含名称、属性和操作，如图2-5所示。

【第23章讨论主动类。】

第六，构件 （component）是系统设计的模块化部件，将实现隐藏在一组外部接口背后。在一个系统中，共享相同接口的构件可以相互替换，只要保持相同的逻辑行为即可。可以通过把部件和连接件接合在一起表示构件的实现；部件可以包括更小的构件。在图形上，构件的表示很像类，只是在其右上角有一个特殊的图标，如图2-6所示。

【第15章讨论构件和内部结构。】

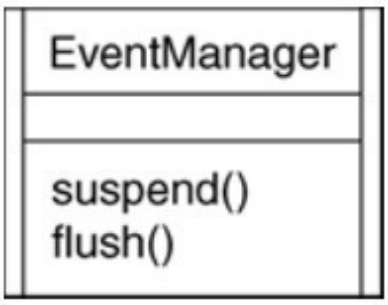


图2-5 主动类

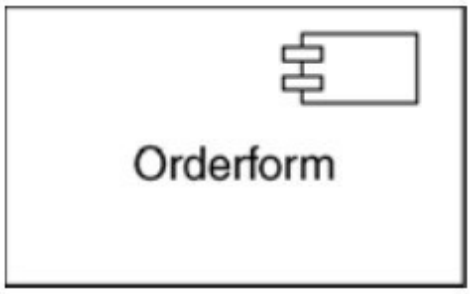


图2-6 构件

剩下的两种元素是制品和结点，它们也是不同的。它们表示物理事物，而前6种元素表示概念或逻辑事物。

第七，制品（artifact）是系统中物理的而且可替换的部件，它包括物理信息（“比特”）。在一个系统中，会遇到不同类型的部署制品，如源代码文件、可执行程序和脚本。制品通常代表对源码信息或运行时信息的物理打包。在图形上，把制品画成一个矩形，在其名称的上方标注着关键字«artifact»，如图2-7所示。

【第26章讨论制品。】

第八，结点（node）是在运行时存在的物理元素，它表示一个计算机资源，通常至少有一些记忆能力，还经常具有处理能力。一组构件可以驻留在一个结点内，也可以从一个结点迁移到另一个结点。在图形上，把结点画成一个立方体，通常在立方体中只写它的名称，如图2-8所示。

【第27章讨论结点。】

图2-7 制品

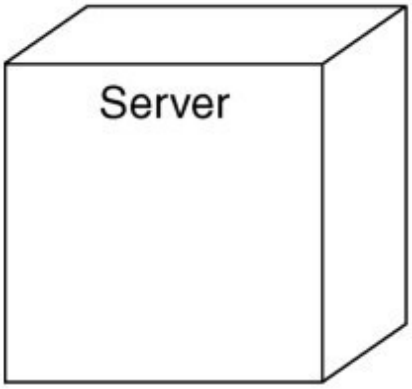


图2-8 结点

这些元素——类、接口、协作、用况、主动类、构件、制品和结点，是UML模型中可以包含的基本结构事物。它们也有变体，如参与者、信号、实用程序（几种类）、进程和线程（两种主动类）、应用、文档、文件、库、页和表（几种制品）等。

3.行为事物

行为事物 behavioral thing）是UML模型的动态部分。它们是模型中的动词，代表了跨越时间和空间的行为。共有3类主要的行为事物。

第一，交互 （interaction）是这样一种行为，它由在特定语境中共同完成一定任务的一组对象或角色之间交换的消息组成。一个对象群体的行为或者单个操作的行为可以用一个交互来描述。交互涉及一些其他元素，包括消息、动作和连接件（对象间的连接）。在图形上，把消息画成一条有方向的直线，通常在其上总是带有操作名，如图2-9所示。

图2-9 消息

【第17章讨论用于模型中构造行为事物的用况，在第16章讨论交互。】

第二，状态机 （state machine）是这样一种行为，它描述了一个对象或一个交互在生命期内响应事件所经历的状态序列以及它对这些事件做出的响应。单个类或一组类之间协作的行为可以用一个状态机来描述。状态机涉及到一些其他元素，包括状态、转移（从一个状态到另一个状态的流）、事件（触发转换的事物）和活动（对一个转移的响应）。在图形上，把状态画成一个圆角矩形，通常在其中含有状态的名字及其子状态（如果有的话），如图2-10所示。

【第22章讨论状态机。】

第三，活动 （activity）是这样一种行为，它描述了计算过程执行的步骤序列。交互所注重的是一组进行交互的对象，状态机所注重的是一定时间内一个对象的生命周期，活动所注重的是步骤之间的流而不关心哪个对象执行哪个步骤。活动的一个步骤称为一个动作。在图形上，把动作画成一个圆角矩形，在其中含有指明其用途的名字，如图2-11 所示。状态和动作靠不同的语境得以区别。

图2-10 状态

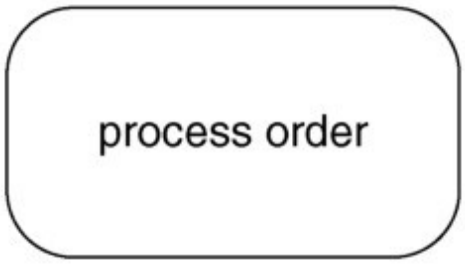


图2-11 动作

交互、状态机和活动这3种元素是UML模型中可能包含的基本行为事物。在语义上，这些元素通常与各种结构元素（主要是类、协作和对象）相关。

4.分组事物

分组事物 （grouping thing）是UML模型的组织部分。它们是一些由模型分解成的“盒子”。主要的分组事物是包。

包 （package）是用于对设计本身进行组织的通用机制，与类不同，它是用来组织实现构造物的。结构事物、行为事物甚至其他的分组事物都可以放进包内。包不像构件（构件在运行时存在），它纯粹

是概念上的（即它仅在开发时存在）。在图形上，把包画成带标签的文件夹（一个左上角带有一个小矩形的大矩形），在矩形中通常仅含有包的名称，有时还含有其内容，如图2-12所示。

【第12章讨论包。】

包是用来组织UML模型的基本分组事物。它也有变体，如框架、模型和子系统（它们是包的不同种类）。

5.注释事物

注释事物 （annotational thing）是UML模型的解释部分。这些注释事物用来描述、说明和标注模型中的任何元素。有一种主要的注释事物，称为注解。注解（note）是依附于一个元素或一组元素之上对它进行约束或解释的简单符号。在图形上，把注解画成一个右上角是折角的矩形，其中带有文字或图形解释，如图2-13所示。

【第6章讨论注解。】

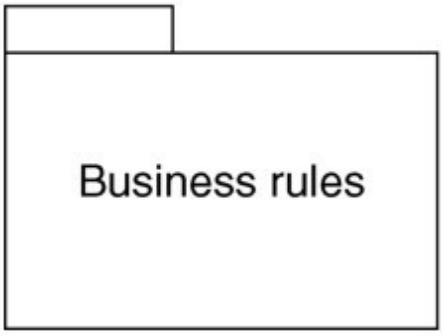


图2-12 包

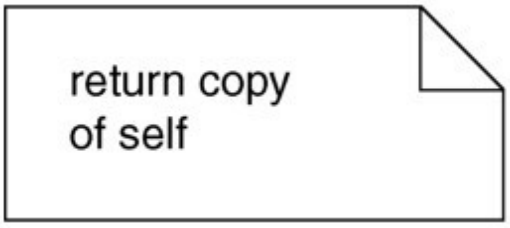


图2-13 注解

该元素是可以包含在UML模型中的基本注释事物。通常可以用注解中所含的约束或解释来修饰图，最好是把注释表示成形式或非形式

化的文本。这种元素也有变体，例如需求（从模型的外部来描述一些想得到的行为）。

6.UML中的关系

在UML中有4种关系：

（1）依赖；

（2）关联；

（3）泛化；

（4）实现。

这些关系是UML的基本关系构造块，用它们可以写出形式良好的模型。

第一，依赖 （dependency）是两个模型元素间的语义关系，其中一个元素（独立元素）发生变化会影响另一个元素（依赖元素）的语义。在图形上，把依赖画成一条可能有方向的 [2] 虚线，有时还带有一个标记，如图2-14所示。

【第5章和第10章讨论依赖。】

第二，关联 （association）是类之间的结构关系，它描述了一组链，链是对象（类的实例）之间的连接。聚合是一种特殊类型的关联，它描述了整体和部分间的结构关系。在图形上，把关联画成一条实线，它可能有方向，有时还带有一个标记，而且它还经常含有诸如多重性和端名这样的修饰，如图2-15所示。

【第5章和第10章讨论关联。】

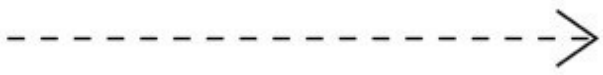


图2-14 依赖

图2-15 关联

第三，泛化 （generalization）是一种特殊/一般关系，其中特殊元素（子元素）基于一般元素（父元素）而建立。用这种方法，子元素共享了父元素的结构和行为。在图形上，把泛化关系画成一条带有空心箭头的实线，该实线指向父元素，如图2-16所示。

【第5章和第10章讨论泛化。】

第四，实现 （realization）是类目之间的语义关系，其中一个类目指定了由另一个类目保证执行的合约。在两种地方会遇到实现关系：一种是在接口和实现它们的类或构件之间；另一种是在用况和实现它们的协作之间。在图形上，把实现关系画成一条带有空心箭头的虚线，它是泛化和依赖关系两种图形的结合，如图2-17所示。

【第10章讨论实现关系。】

图2-16 泛化

图2-17 实现

这4种元素是UML模型中可以包含的基本关系事物。它们也有变体，例如，精化、跟踪、包含和扩展。

7.UML中的图

图（diagram）是一组元素的图形表示，大多数情况下把图画成顶点（代表事物）和弧（代表关系）的连通图。为了对系统进行可视化，可以从不同的角度画图，这样一个图是对系统的投影。对所有的系统（除非很微小的系统）而言，图是系统组成元素的省略视图。有些元素可以出现在所有图中，有些元素可以出现在一些图中（很常见），还有些元素不能出现在图中（很罕见）。在理论上，图可以包含事物及其关系的任何组合。然而在实际中仅出现少量的常见组合，

它们与组成软件密集型系统的体系结构的5种最有用的视图相一致。由于这个原因，UML包括13种这样的图：

【在本章后面讨论体系结构的5种视图。】

（1）类图；
（2）对象图；
（3）构件图；
（4）组合结构图；
（5）用况图；
（6）顺序图 [3] ；
（7）通信图；
（8）状态图；
（9）活动图；
（10）部署图；
（11）包图；
（12）定时图；
（13）交互概览图。

类图（class diagram）展现了一组类、接口、协作和它们之间的关系。在面向对象系统的建模中所建立的最常见的图就是类图。类图给出系统的静态设计视图。包含主动类的类图给出系统的静态进程视图。构件图是类图的变体。

【第8章讨论类图。】

对象图（object diagram）展现了一组对象以及它们之间的关系。对象图描述了在类图中所建立的事物的实例的静态快照。和类图一样，这些图给出系统的静态设计视图或静态进程视图，但它们是从真实案例或原型案例的角度建立的。

【第14章讨论对象图。】

构件图 （component diagram）展现了一个封装的类和它的接口、端口以及由内嵌的构件和连接件构成的内部结构。构件图用于表示系统的静态设计实现视图。对于由小的部件构建大的系统来说，构件图是很重要的（UML 将构件图和适用于任意类的组合结构图区分开来，但由于构件和结构化类之间的差别微不足道，所以一起讨论它们）。

【第15章讨论构件图和内部结构。】

用况图 （use case diagram）展现了一组用况、参与者（一种特殊的类）及它们之间的关系。用况图给出系统的静态用况视图。这些图在对系统的行为进行组织和建模上是非常重要的。

【第18章讨论用况图。】

顺序图和通信图都是交互图。交互图（interaction diagram）展现了一种交互，它由一组对象或角色以及它们之间可能发送的消息构成。交互图专注于系统的动态视图。顺序图（sequence diagram）是强调消息的时间次序的交互图；通信图（communication diagram）也是一种交互图，它强调收发消息的对象或角色的结构组织。顺序图和通信图表达了类似的基本概念，但每种图强调概念的不同视角，顺序图强调时间次序，通信图强调消息流经的数据结构。定时图（不包含在本书中）展现了消息交换的实际时间。

【第19章讨论交互图。】

状态图 （state diagram）展现了一个状态机，它由状态、转移、事件和活动组成。状态图展现了对象的动态视图。它对于接口、类或协作的行为建模尤为重要，而且它强调由事件引发的对象行为，这非常有助于对反应式系统建模。

【第25章讨论状态图。】

活动图 （activity diagram）将进程或其他计算的结构展示为计算内部一步一步的控制流和数据流。活动图专注于系统的动态视图。它对于系统的功能建模特别重要，并强调对象间的控制流程。

【第20章讨论活动图。】

部署图 （deployment diagram）展现了对运行时的处理结点以及在其中生存的构件的配置。部署图给出了体系结构的静态部署视图。通常一个结点包含一个或多个制品。

【第31章讨论部署图。】

制品图 （artifact diagram）展现了计算机中一个系统的物理结构。制品包括文件、数据库和类似的物理比特集合。制品常与部署图一起使用。制品也展现了它们实现的类和构件。（UML把制品图视为部署图的变体，但我们分别地讨论它们。）

【第30章讨论制品图。】

包图 （package diagram）展现了由模型本身分解而成的组织单元以及它们的依赖关系。

【第12章讨论包图。】

定时图 （timing diagram）是一种交互图，它展现了消息跨越不同对象或角色的实际时间，而不仅仅是关心消息的相对顺序。交互概览图 （interaction overview diagram）是活动图和顺序图的混合物。这些图有特殊的用法，本书不做讨论，更多的细节可参考The Unified Modeling Language Reference Manual。

并不限定仅使用这几种图，开发工具可以利用UML来提供其他种类的图，但到目前为止，这几种图在实际应用中是最常用的。

2.2.2 UML规则

不能简单地把 UML 的构造块按随机的方式堆放在一起。像任何语言一样，UML 有一套规则，这些规则描述了一个形式良好的模型应该是什么样。形式良好的模型应该在语义上是自我一致的，并且与所有的相关模型协调一致。

UML有自己的语法和语义规则，用于:

命名——为事物、关系和图起的名字；

范围——使名字具有特定含义的语境；

可见性——这些名字如何让其他成分看见和使用；

完整性——事物如何正确、一致地相互联系；

执行——运行或模拟一个动态模型意味着什么。

在软件密集型系统的开发期间所建造的模型往往需要发展变化，并可以由许多人员以不同的方式、在不同的时间进行观察。由于这个原因，下述的情况是常见的，即开发组不但会建造一些形式良好的模型，也会建造一些像下面这样的模型：

省略——隐藏某些元素以简化视图；

不完全——可能遗漏了某些元素；

不一致——模型的完整性得不到保证。

在软件开发的生命期内，随着系统细节的展开和变动，不可避免地要出现这样一些不太规范的模型。UML 的规则鼓励（不是强迫）专注于最重要的分析、设计和实现问题，这将促使模型随着时间的推移而具有良好的结构。

2.2.3 UML中的公共机制

通过与具有公共特征的模式取得一致，可以使一座建筑更为简单和更为协调。房子可以按一定的结构模式（它定义了建筑风格）建造成维多利亚式的或法国乡村式的。对于UML也是如此。由于在UML中有4种贯穿整个语言且一致应用的公共机制，因此使得UML变得较为简单。这4种机制是：

（1）规约；

（2）修饰；

（3）通用划分；

（4）扩展机制。

1.规约

UML 不仅仅是一种图形语言。实际上，在它的图形表示法的每部分背后都有一个规约，这个规约提供了对构造块的语法和语义的文字叙述。例如，在类的图符背后有一个规约，它提供了对该类所拥有的属性、操作（包括完整的特征标记）和行为的全面描述；在视觉上，类的图符可能仅展示了这个规约的一小部分。此外，可能存在着该类的另一个视图，其中提供了一个完全不同的部件集合，但是它仍然与该类的基本规约相一致。UML 的图形表示法用来对系统进行可视化；UML 的规约用来说明系统的细节。假定把二者分开，就可能进行增量式的建模。这可以通过以下方式完成：先画图，然后再对这个模型的规约增加语义，或直接创建规约，也可能对一个已经存在的系统进行逆向工程，然后再创建作为这些规约的投影的图。

UML 的规约提供了一个语义底版，它包含了一个系统的各个模型的所有部分，各部分以一致的方式相互联系。因此，UML 的图只不过是对底版的简单视觉投影，每一个图展现了系统的一个特定的关注方面。

2.修饰

UML中的大多数元素都有唯一而直接的图形表示符号，这些图形符号对元素的最重要的方面提供了可视化表示。例如，特意把类的符号设计得容易画出，这是因为在面向对象系统建模中，类是最常用的元素；类的图形符号展示了类的最重要方面，即它的名称、属性和操作。

【第6章讨论注解和其他修饰。】

对类的规约可以包含其他细节，例如，它是否为抽象类，或它的属性和操作是否可见。可以把很多这样的细节表示为图形或文字修饰，放到类的基本矩形符号上。例如，图2-18 表示的是一个带有修饰

的类，图中表明这个类是一个抽象类，有两个公共操作、一个受保护操作和一个私有操作。

UML表示法中的每一个元素都有一个基本符号，可以把各种修饰细节加到这个符号上。

3.通用划分

在对面向对象系统建模中，通常有几种划分方式。

第一种方式是对类和对象的划分。类是一种抽象，对象是这种抽象的一个具体表现。在UML中，可以对类和对象建立模型，如图2-19所示。在图形上，UML是这样区分对象的：采用与类同样的图形符号来表示对象，并且在对象名的下面画一道线。

【第13章中讨论对象。】

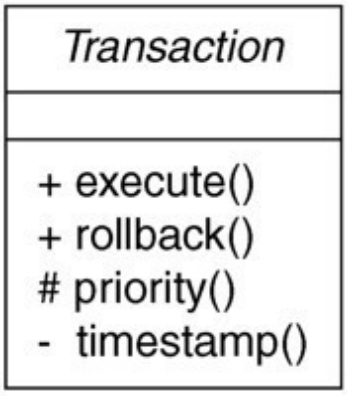


图2-18 修饰

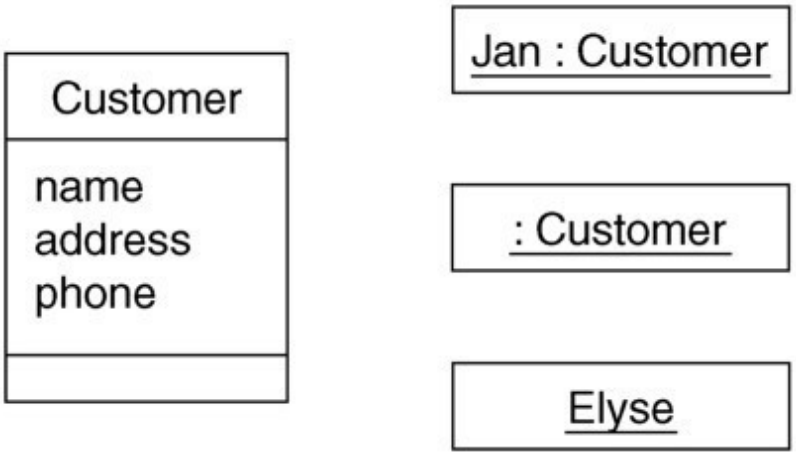


图2-19 类和对象

在这个图中，有一个名称为Customer的类，它有3个对象，分别为Jan（它被明确地标记为Customer的对象），:Customer（匿名的Customer对象）和Elyse（它在规约中被说明为一种Customer对象，尽管在这里没有明确地表示出来）。

UML的每一个构造块几乎都存在像类/对象这样的二分法。例如，可以有用况和用况执行、构件和构件实例、结点和结点实例等。

第二种方式是接口和实现的分离。接口声明了一个合约，而实现则表示了对该合约的具体实施，它负责如实地实现接口的完整语义。在UML中，既可以对接口建模又可以对它们的实现建模，如图2-20所示。

【第11章讨论接口。】

图2-20 接口和实现

在这个图中，有一个名称为SpellingWizard.dll的构件，它实现了接口IUnknown和接口ISpelling，并且还需要一个由其他构件提供的名为IDictionary的接口。

几乎每一个UML的构造块都有像接口/实现这样的二分法。例如，用况和实现它们的协作，操作和实现它们的方法。

第三种方式是类型和角色的分离。类型声明了实体的种类（如对象、属性或参数），角色描述了实体在语境中的含义（如类、构件或

协作等）。任何作为其他实体结构中的一部分的实体（例如属性）都具有两个特性：从它固有的类型派生出一些含义，从它在语境中的角色派生出一些含义（如图2-21所示）。

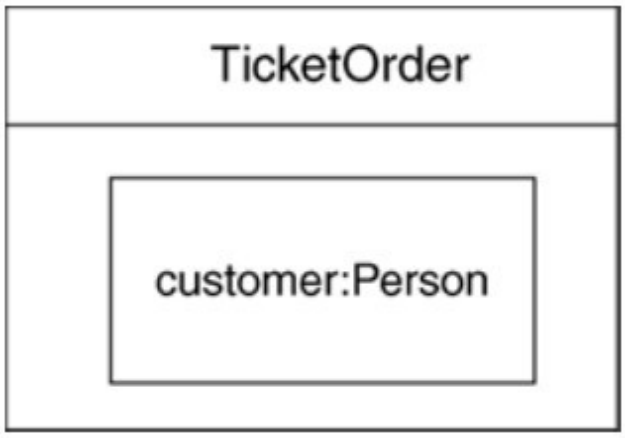


图2-21 具有角色和类型的部件

4.扩展机制

UML 提供了一种绘制软件蓝图的标准语言，但是一种闭合的语言即使表达能力再丰富，也难以表示出各种领域中的各种模型在不同时刻所有可能的细微差别。由于这个原因，UML 是目标开放的，使人们能够以受控的方式来扩展该语言。UML的扩展机制包括：

衍型；

标记值；

约束。

衍型 （stereotype）扩展了UML的词汇，可以用来创造新的构造块，这个新构造块既是从现有的构造块派生的，但是针对专门的问题。例如，假设正在使用一种编程语言，如Java或C++，经常要对“异常事件”建模。在这些语言里，“异常事件”就是类，只是用很特殊的方法进行了处理。通常可能只想允许抛出和捕捉异常事件，没有其他要求。此时可以让异常事件在模型中成为“一等公民”——可以像对待基

本构造块一样对待它们，只要用一个适当的衍型来标记它们即可。请看图2-22中的类Overflow。

【第6章讨论UML的扩展机制。】

标记值 （tagged value）扩展了UML衍型的特性，可以用来创建衍型规约的新信息。例如，如果在制作以盒装形式销售的产品，随着时间的推移，它经过了多次发行，那么经常会想要跟踪产品的版本和对产品做关键摘要的作者。版本和作者不是UML的基本概念，通过引入新的标记值，可以把它们加到像类那样的任何构造块中去。例如，在图2-22中，在类EventQueue上明确标记了版本和作者，这样就对该类进行了扩展。

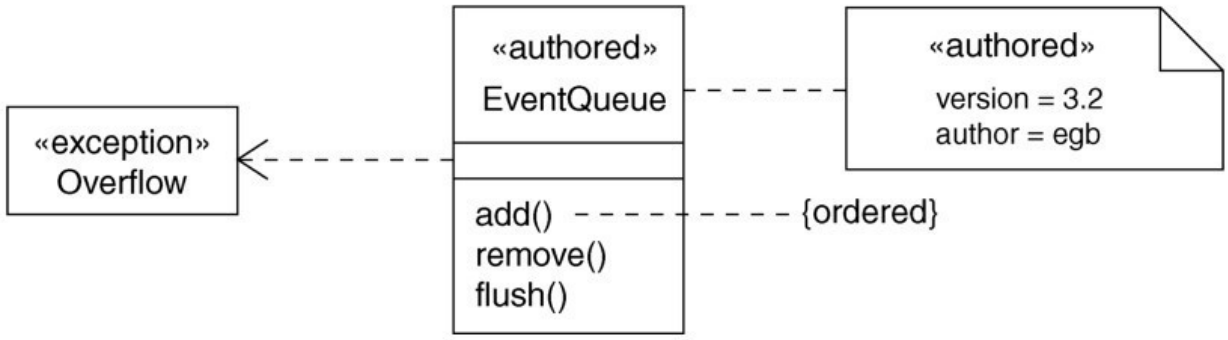


图2-22 扩展机制

约束 （constraint）扩展了UML构造块的语义，可以用来增加新的规则或修改现有的规则。例如，可能想约束类 EventQueue，以使所有的增加都按序排列。如图2-22 所示，对操作 add增加了一个约束，即{ordered}，以明确标示这一规则。

总的来说，这3种扩展机制允许根据项目的需要来塑造和培育UML。这些机制也使得UML适合于新的软件技术（例如，很可能出现的功能更强的分布式编程语言）。可以增加新的构造块，修改已存在的构造块的规约，甚至可以改变它们的语义。当然，以受控的方式进行扩展是重要的，这样可以不偏离UML的目标——信息交流。

2.3 体系结构

可视化、详述、构造和文档化一个软件密集型系统，要求从几个角度去观察系统。各种人员——最终用户、分析人员、开发人员、系统集成人员、测试人员、技术资料作者和项目管理者——各自带着项目的不同日程，在项目的生命周期内各自在不同的时间、以不同的方式来看系统。系统体系结构或许是最重要的制品，它可以驾驭不同的视点，并在整个项目的生命周期内控制对系统的迭代和增量式开发。

【第1章讨论需要从不同的角度观察复杂系统。】

体系结构是一组有关下述内容的重要决策：

软件系统的组织；

对组成系统的结构元素及其接口的选择；

像元素间的协作所描述的那样的行为；

将这些结构元素和行为元素组合到逐步增大的子系统中；

指导这种组织的体系结构风格：静态和动态元素以及它们的接口、协作和组成。

软件体系结构不仅关心结构和行为，而且还关心用法、功能、性能、弹性、复用、可理解性、经济与技术约束及其折中，以及审美的考虑。

如图2-23所示，最好用5个互连的视图来描述软件密集型系统的体系结构。每一个视图是在一个特定的方面对系统的组织和结构进行的投影。

【第32章讨论对系统的体系结构建模。】

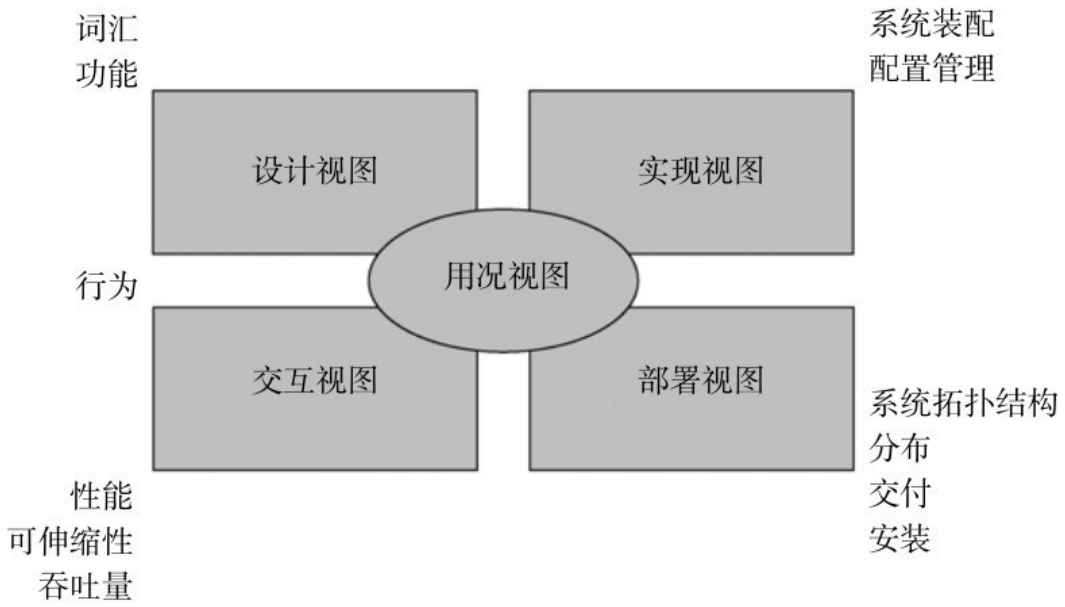


图2-23 对系统的体系结构建模

系统的用况视图 （use case view）由描述可被最终用户、分析人员和测试人员看到的系统行为的用况组成。用况视图实际上没有描述软件系统的组织，而是描述了形成系统体系结构的动力。在UML中，该视图的静态方面由用况图表现；动态方面由交互图、状态图和活动图表现。

系统的设计视图 （design view）包含了类、接口和协作，它们形成了问题及其解决方案的词汇。这种视图主要支持系统的功能需求，即系统应该提供给最终用户的服务。在UML中，该视图的静态方面由类图和对象图表现；动态方面由交互图、状态图和活动图表现。类的内部结构图特别有用。

系统的交互视图 （interaction view）展示了系统的不同部分之间的控制流，包括可能的并发和同步机制。该视图主要针对性能、可伸缩性和系统的吞吐量。在UML中，对该视图的静态方面和动态方面的表现与设计视图相同，但着重于控制系统的主动类和在它们之间流动的消息。

系统的实现视图 （implementation view）包含了用于装配与发布物理系统的制品。这种视图主要针对系统发布的配置管理，它由一些独立的文件组成；这些文件可以用各种方法装配，以产生运行系统。它也关注从逻辑的类和构件到物理制品的映射。在UML中，该视图的静态方面由构件图表现，动态方面由交互图、状态图和活动图表现。

系统的部署视图 （deployment view）包含了形成系统硬件拓扑结构的结点（系统在其上运行）。这种视图主要描述组成物理系统的部件的分布、交付和安装。在UML中，该视图的静态方面由部署图表现，动态方面由交互图、状态图和活动图表现。

这5种视图中的每一种都可单独使用，使不同的人员能专注于他们最为关心的体系结构问题。这5种视图也会相互作用，如部署视图中的结点拥有实现视图的构件，而这些构件又表示了设计视图和交互视图中的类、接口、协作以及主动类的物理实现。UML允许表达这5种视图中的任何一种。

2.4 软件开发生命周期

UML 在很大程度上是独立于过程的，这意味着它不依赖于任何特殊的软件开发生命周期。然而，为了从UML中得到最大的收益，应该考虑这样的过程，它是：

用况驱动的；

以体系结构为中心的；

迭代的和增量的。

【在附录B中概述了Rational统一过程，对该过程的更完整处理在The Unified Software Development Process一书以及The Rational Unified Process中讨论。】

用况驱动 （use case driven）意味着把用况作为一种基本的制品，用于建立所要求的系统行为、验证和确认系统的体系结构、测试以及在项目组成员间进行交流。

以体系结构为中心 （architecture-centric）意味着以系统的体系结构作为一种基本制品，对被开发的系统进行概念化、构造、管理和演化。

迭代过程 （iterative process）是这样一种过程，它涉及到对一连串可执行的发布的管理。增量过程 （incremental process）是这样一种过程，它涉及到系统体系结构的持续集成，以产生各种发布，每个新的发布都比上一个发布有所改善。总的来讲，迭代和增量的过程是风险驱动的（risk-driven），这意味着每个新的发布都致力于处理和降低对于项目成功影响最为显著的风险。

这种用况驱动的、以体系结构为中心的、迭代/增量的过程可以分成几个阶段。阶段 （phase）是过程的两个主要里程碑之间的时间跨度，在阶段中将达到一组明确的目标，完成一定的制品，并做出是否进入到下一阶段的决策。如图2-24所示，在软件开发生命周期内有4个阶段：初始、细化、构造和移交。在图中，按这些阶段对工作流进行了划分，并显示了它们的焦点随时间的推移而变化的程度。

初始 （inception）是这个过程的第一个阶段。在此阶段，萌发的开发想法经过培育要达到这样一个目标：至少要在内部奠定足够的基础，以保证能够进入到细化阶段。

细化 （elaboration）是这个过程的第二个阶段。在此阶段定义产品需求和体系结构。在这个阶段，将明确系统需求，按其重要性排序并划定基线。可以按一般的描述，也可以按精确的评价准则来排列系统的需求，每个需求都说明了特定的功能或非功能的行为，并为测试提供了基础。

构造 （construction）是这个过程的第三个阶段，在此阶段软件从可执行的体系结构基线发展到准备移交给用户。针对项目的商业需要，这里也要不断地对系统的需求，特别是对系统的评价准则进行检查，并要适当地分配资源，以主动地降低项目的风险。

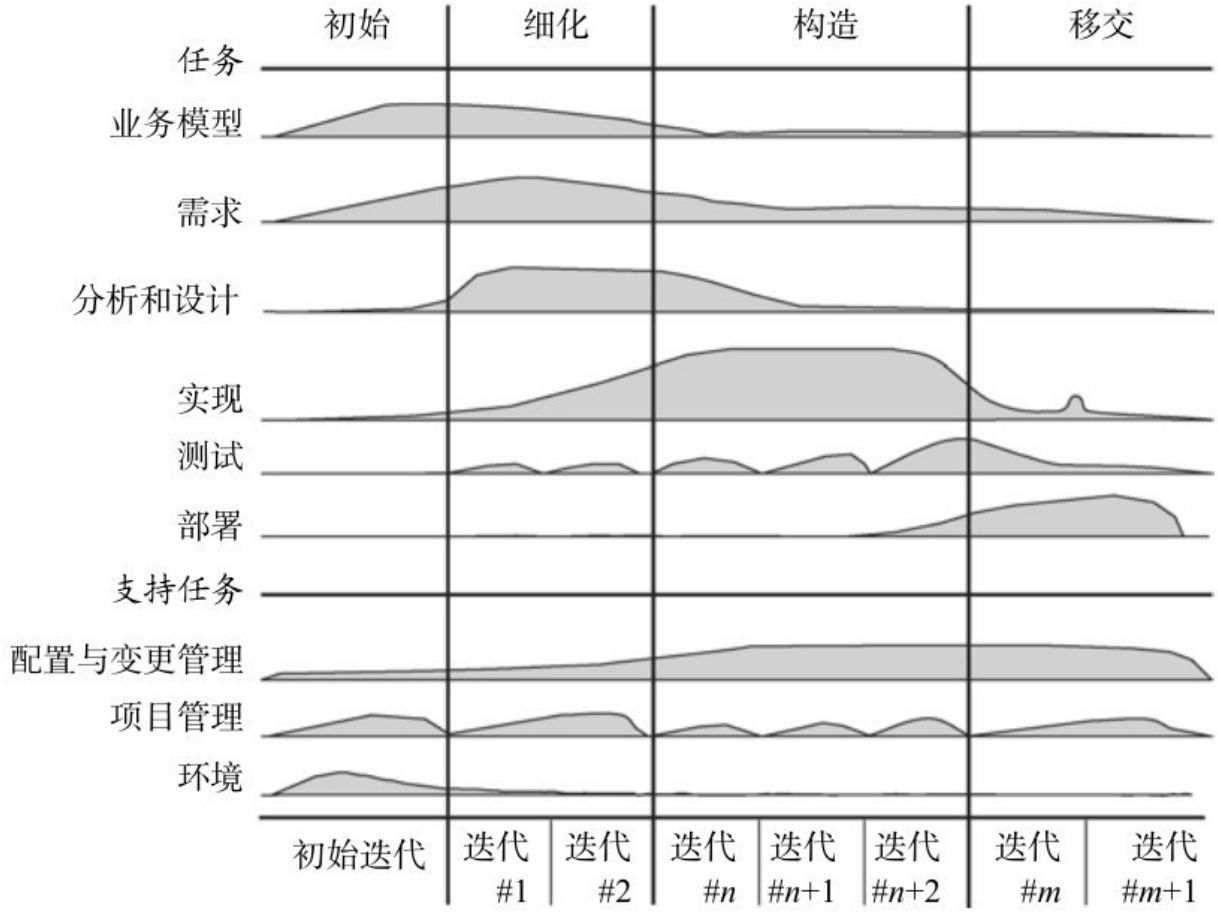


图2-24 软件开发生命周期

移交 （transition）是这个过程的第四个阶段，在此阶段把软件交付给用户。在这个阶段，软件开发过程很少能结束，还要继续改善系统，根除错误，增加早期发布未能实现的特性。

使得这个过程与众不同，并贯穿所有 4个阶段的要素是迭代。迭代（iteration）是一组明确的工作任务，具有产生能运行、测试和评价的可执行系统的基准计划和评价准则。可执行系统无须向外发布。因为

迭代产生可执行的产品，所以可以判断进展并在每次迭代后可重新估计风险。这意味着，软件开发生命周期具有以下特征：持续地发布系统体系结构的可执行版本，而且在每步迭代后可中途进行修改，以减少潜在的风险。正是因为强调将体系结构作为一个重要的制品，UML非常注重对系统体系结构的不同视图进行建模。

第3章 Hello,World！

本章内容

类和制品

静态模型和动态模型

模型间的联系

对UML的扩展

C编程语言的发明者Brian Kernighan和Dennis Ritchie指出，学习一门新的编程语言的唯一方法是用它编写程序。对于UML也是如此，学习UML的唯一方法是用它绘制模型。

当开始学习一门新的编程语言时，很多开发者写的第一个程序是简单的，只包含了一些打印“Hello,World!”字符串之类的语句。这是一个合理的出发点，因为掌握这种小应用可以立见成效。同时，它也覆盖了使某些东西运行所需要的全部基础设施。

我们就从这里开始使用UML。对“Hello,World!”的建模大概是UML最简单的应用。然而，这个应用的简单只是一种假象，因为在整个应用的下面有一些耐人寻味的工作机制。用UML可以很容易地对这些机制建模，UML对这种简单的应用提供了较为丰富的视图。

3.1 关键抽象

在Web浏览器中，打印“Hello,World!”的Java程序是很简单的：

```
import java.awt.Graphics;
class HelloWorld extends java.applet.Applet {
   public void paint (Graphics g) {
      g.drawString("Hello,World!",10,10);
   }
}
```

第一行代码

```
import java.awt.Graphics;
```

使得后面的代码可以直接使用类 Graphics。前缀 java.awt 表明了类 Graphics 所在的

Java包。

第二行代码

```
class HelloWorld extends java.applet.Applet {
```

介绍了一个名为HelloWorld 的新类，并说明它是一个像 Applet 那样的类，Applet 位于包java.applet中。

其余的三行代码

```
public void paint (Graphics g) {
   g.drawString("Hello,World!",10,10);
}
```

声明了一个名为paint的操作，它的实现调用名为drawString的另一个操作，drawString操作负责在指定的位置上打印“Hello,World!”。在通常的面向对象的方式下，drawString是一个名称为g的参数上的一个操作，g的类型是类Graphics。

在UML中，对这种应用的建模是简单的。如图3-1所示，把类HelloWorld用一个矩形图标表示。类HelloWorld的paint操作也展示在这

里（省略了该操作的所有形式参数），在一个附属的注解中详述了该操作的实现。

【第4章和第9章讨论类。】

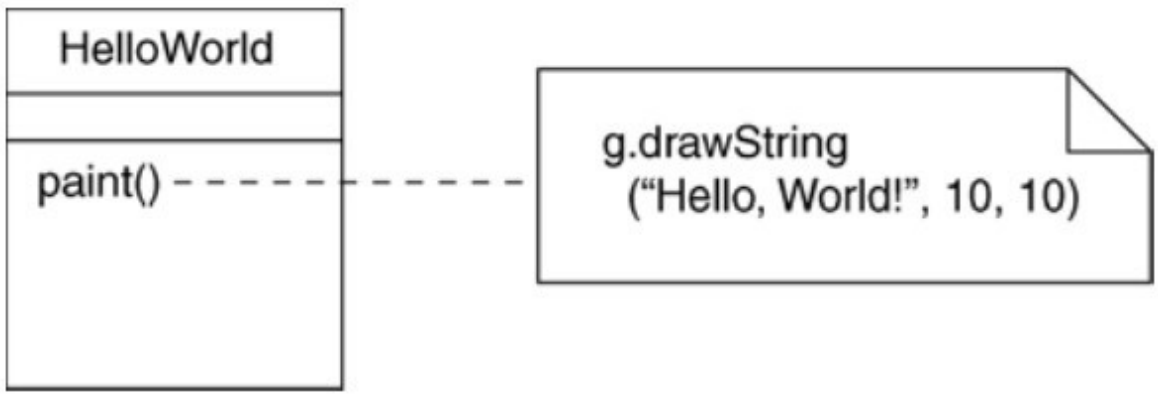


图3-1 对HelloWorld的关键抽象

注解 如图3-1 中所示的那样，虽然 UML 允许（但不要求）紧密地与各种编程语言（如Java）相结合，但UML不是一种可视化的编程语言。UML被设计为允许把模型转换成代码，也允许把代码通过逆向工程转换为模型。在 UML 中，像数学表达式这样的事物最好用文字性的编程语言语法来书写，而像类层次这样的事物最好以图形的方式来可视化。

这个类图反映出了“Hello,World!”这个应用的基本部分，但还遗漏了一些东西。按上述代码的描述，这个应用还涉及其他两个类，即Applet和Graphics，而且对每个类的使用方式各不相同。类Applet被用作类HelloWorld的父类，类Graphics则用于类HelloWorld的一个操作paint的特征标记和实现中。可以在类图中表示这些类及它们与类HelloWorld的不同关系，如图3-2所示。

用矩形图标表示类Applet和类Graphics。因为不显示它们的任何操作，所以对它们的图标进行了省略。从HelloWorld到Applet的带有空心箭头的有向线段表示的是泛化关系，在这里它意味着HelloWorld是

Applet的子类。从HelloWorld到Graphics的有向虚线表示的是依赖关系，它意味着HelloWorld使用Graphics。

【第5章和第10章讨论关系。】

至此，HelloWorld建造于其上的框架还没有完工。如果研究Applet和Graphics这两个Java库，将会发现这两个类是一个更大的类层次的一部分。跟踪被类Applet扩展和实现的那些类，能够产生另一个类图，如图3-3所示。

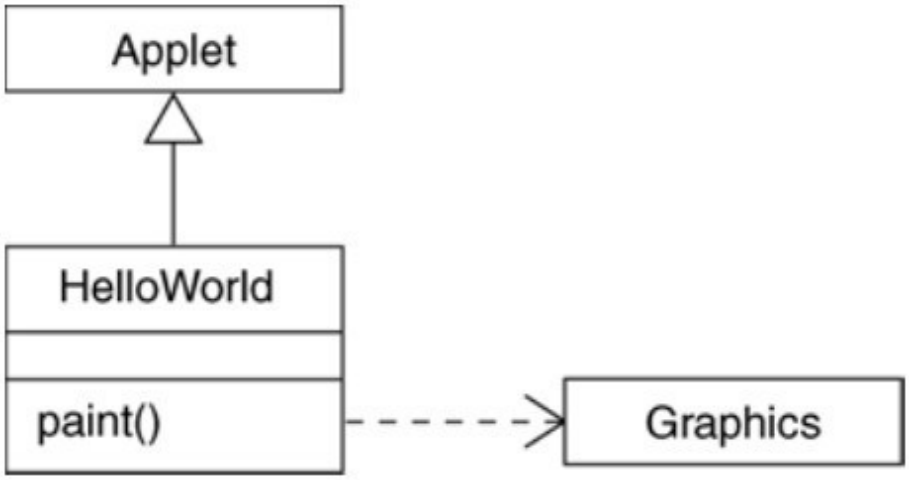


图3-2 与HelloWorld直接相关的类

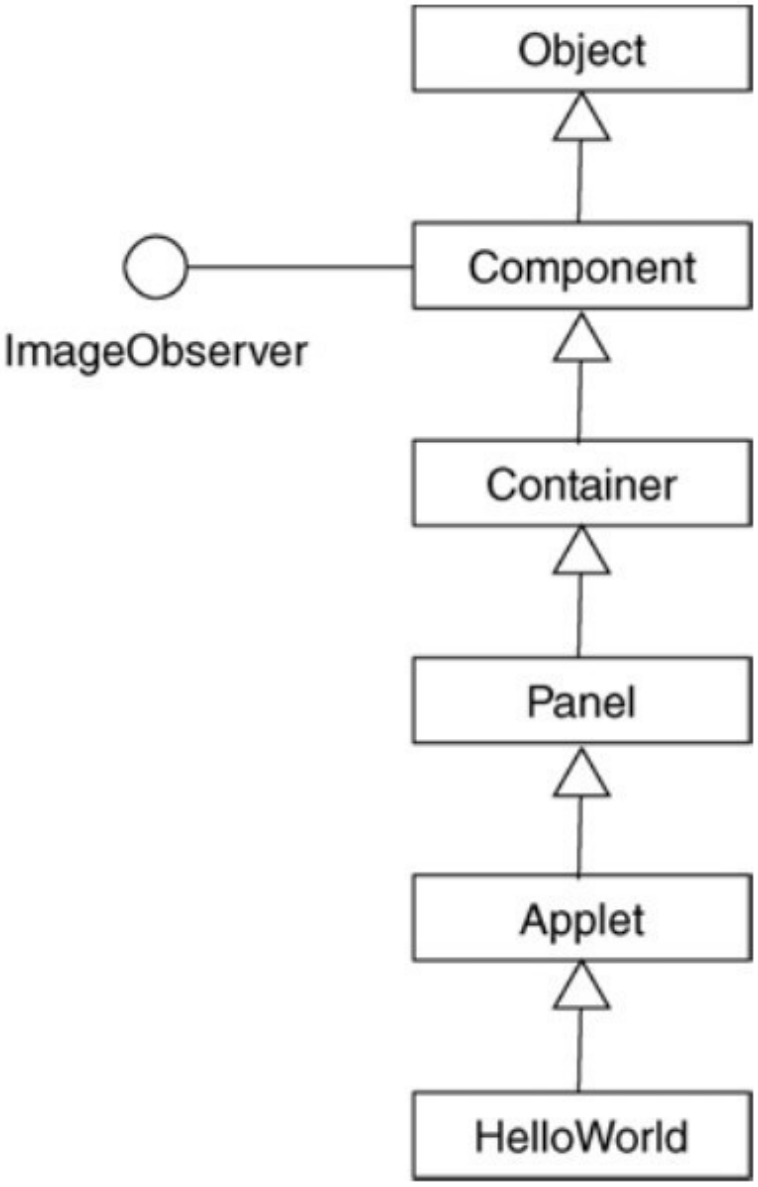


图3-3 HelloWorld的继承层次

注解 通过对一个已经存在的系统进行逆向工程可以产生图，图3-3就是一个很好的例子。逆向工程是从代码创建模型。

可以清楚地从图3-3中看出，HelloWorld仅是这个较大类层次的一个叶子。Hello World是 Applet的子类，Applet是 Panel的子类，Panel是Container的子类，Container是Component的子类，Component是Object的子类，Object是Java中所有类的父类。因而，这个模型与Java库相匹配——每个子类都扩展了某个父类。

ImageObserver和 Component之间的关系有点不同，这在类图中已经反映出来。在Java库中，ImageObserver是一个接口，这意味着它没有实现，而需要其他类实现它。如图3-3所示，在UML中把接口表示成一个圆圈。类Component实现接口ImageObserver这一关系用一条从矩形（Component）到供接口圆圈（ImageObserver）的实线来表示。

【第11章讨论接口。】

如这些图所示，HelloWorld只与两个类（Applet和 Graphics）直接协作，这两个类只是预定义的 Java 大类库的一小部分。为了管理如此大规模的类层次图，Java 用一些不同的包组织它的接口和类。在 Java 环境中，理所应当地把根包命名为Java。嵌套在这个包中的是几个其他的包，每个包还含有其他的包、接口和类。Object 位于包 lang 中，它的完整路径名为 java.lang.Object 。类似地， Panel 、 Container 和 Component位于包awt中，类Applet位于包applet中。接口ImageObserver位于包image中，依次下去，image位于包awt中，因此该接口的完整路径名是一个相当长的串：java.awt.image.ImageObserver。

可以把这个包的在一个类图中可视化，如图3-4所示。在UML中包被表示成带有标签的文件夹。包可以被嵌套，有向的虚线段描述了包之间的依赖。例如，HelloWorld 依赖包java.applet，java.applet依赖包java.awt。

【第12章讨论包。】

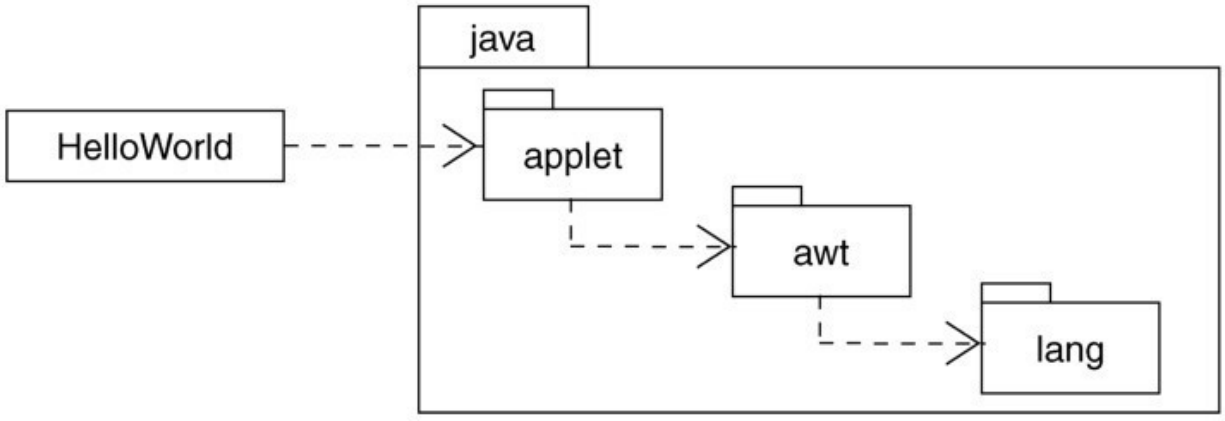


图3-4 HelloWorld依赖的包

3.2 机制

要掌握像 Java 类库这样内容丰富的类库，最难部分是理解各部分如何协同工作。例如，如何调用HelloWorld的操作paint？若想改变这段程序的行为（例如以不同的颜色打印字符串），必须要使用什么样的操作？为了回答这样和那样的问题，必须有一个描述这些类动态协同工作方式的概念模型。

【第29章讨论模式和框架。】

对Java库的研究揭示了HelloWorld的操作paint是从Component继承来的。但是，应该如何调用该操作呢？回答是：paint是在该程序所在的线程运行中调用的，如图3-5所示。

【第23章讨论进程和线程。】

图3-5展示了几个对象间的协作，包括类HelloWorld的一个实例。其他的对象是Java环境的一部分，因此大多数对象都处于所创建的程序的背景中。该图展示了对象之间一个可以被多次应用的协作。每一栏展示了协作中的一个角色，也就是在每次执行中能被不同对象调用的部分。在UML中，把角色表示得像类的样子，只是它们既有角色名也有类型。这个图的中间两个角色是匿名的，因为在协作中它们的类型足以标明它们自身（用冒号并且不加下划线标出它们是角色）。初始的Thread被称为root。角色HelloWorld有一个名称target，这个名称能够被角色ComponentPeer知道。

【第13章讨论实例。】

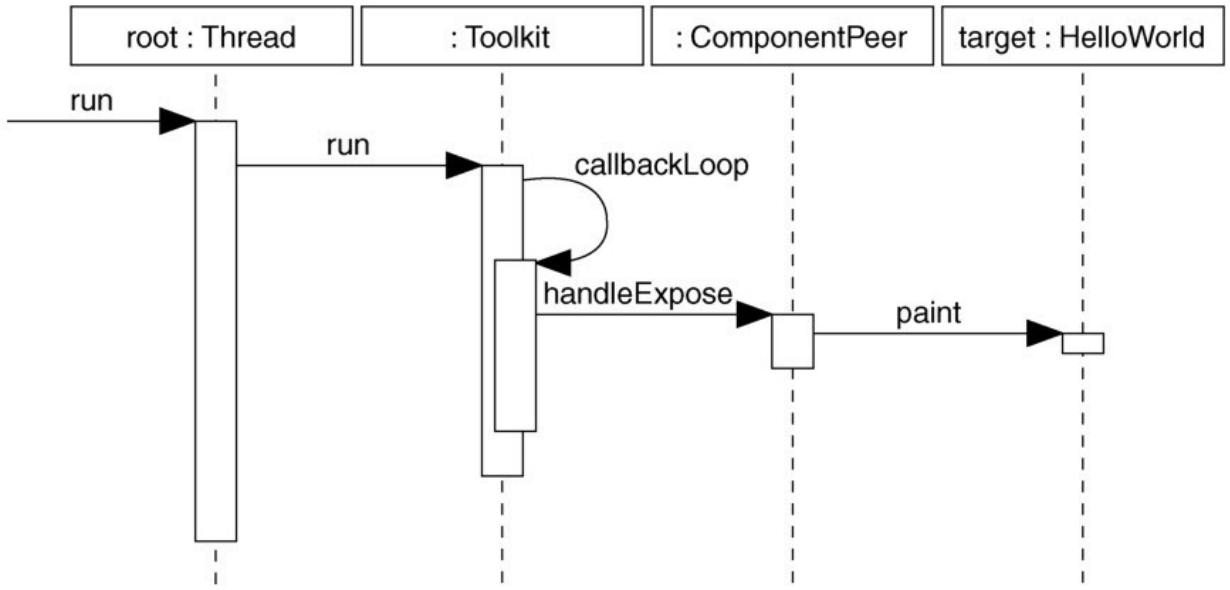


图3-5 paint的工作机制

如图3-5所示，可以使用顺序图对事件的次序建模。这里，序列从运行对象Thread开始，它调用Toolkit的一个操作run。对象Toolkit调用它自己的操作（callbackLoop），然后它调用ComponentPeer的操作handleExpose。对象ComponentPeer调用它的目标操作paint。对象ComponentPeer假设它的目标是Component，但在这种情况下，目标实际是Component的子类（即HelloWorld），因此，HelloWorld的操作paint被多态地处理。

【第19章讨论顺序图。】

3.3 制品

“Hello,World!”被实现为一个小应用程序（applet），它不是单独存在的，而通常是某个网页的一部分。打开含有该程序的网页，并由一些运行该程序的Thread对象的浏览器机制触发，程序就开始执行了。然而，直接作为网页一部分的并不是类HelloWorld，而是由Java编译器（它把描述该类的源代码转换成可执行的制品）产生的该类的二进制

形式。这说明对一个系统存在非常不同的观察角度。所有早先的图描述的是程序的逻辑视图，这里所说的是applet的物理制品的视图。

可以用制品图对这个物理视图建模，如图3-6所示。

【第26章讨论制品。】

逻辑类 HelloWorld 被表示成位于顶部的一个类矩形，这个图中的其他每个图符都表示了系统实现视图中一个UML制品。制品是一种物理表示，如文件。名为hello.java的制品表示了逻辑类HelloWorld的源代码，它是可以由开发环境和配置管理工具操纵的文件。用Java编译器可把这段源代码转换成二进制程序 HelloWorld.class，以使得它适合于在计算机的 Java虚拟机上执行。源代码和二进制程序都表现了（即物理上实现了）逻辑类，这被表示为带着关键字«manifest»的虚线箭头。

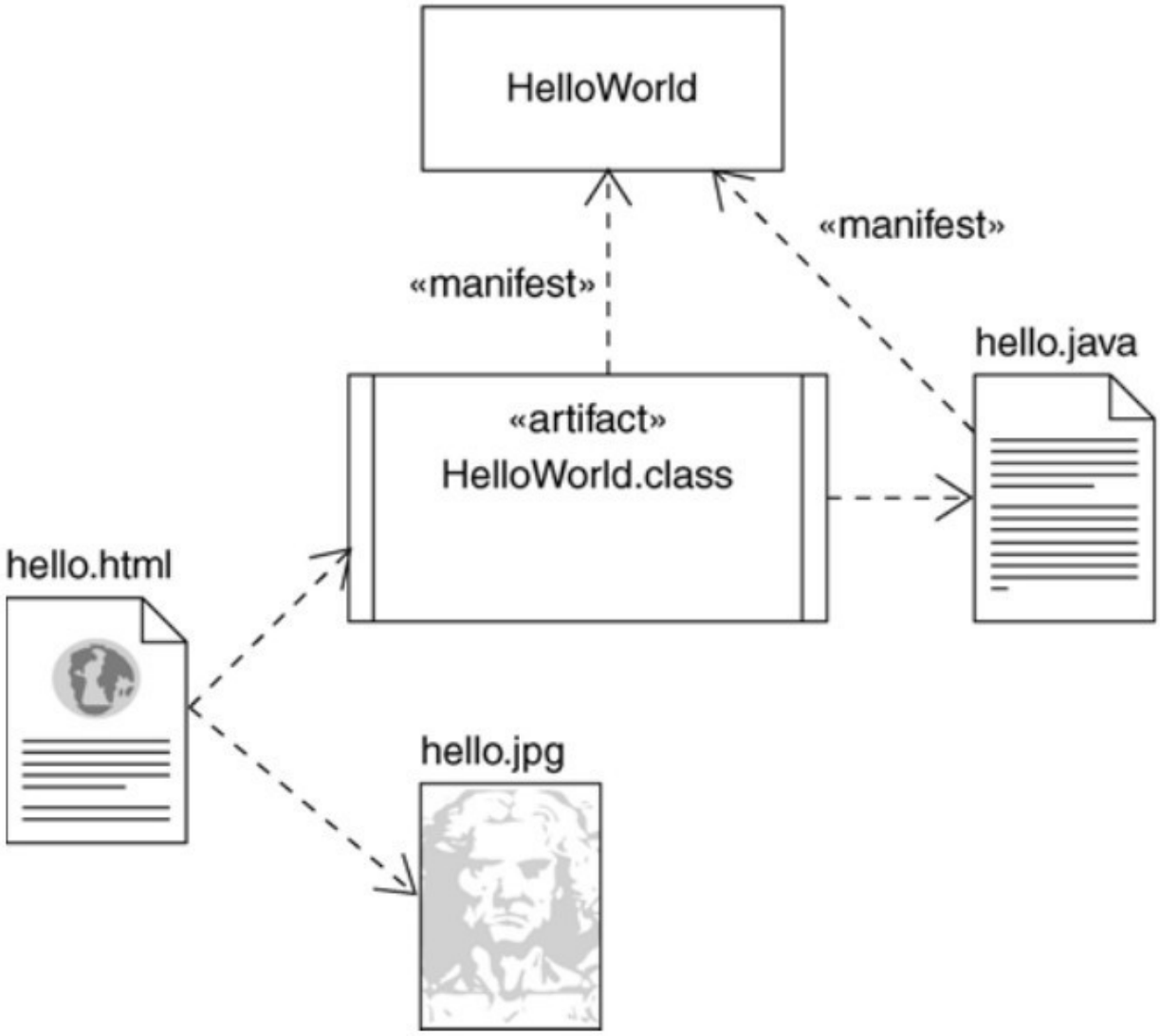


图3-6 HelloWorld制品

制品的图符是名字之上标注了关键字«artifact»的矩形。二进制程序HelloWorld.class是这种基本图符的变体，把图符的线条加粗，是为了指明它是一个可执行的构件（正像主动类一样）。制品 hello.java 的图符已经由用户定义的图符代替，表示它是一个文本文件。网页hello.html的图符也是这样通过扩展UML的表示法来定制的。如图3-6所示，这个网页还有另一个制品 hello.jpg，它是由用户定义的制品图标表示的，这里提供了一个图像的缩影。由于后3个制品使用的是用户定义

的图符，所以它们的名字放在图符外面。制品间的依赖用虚箭头表示。

【第6章讨论UML的扩展机制。】

注解 对类（HelloWorld）、类的源代码（hello.java）和类的目标代码（Hello-World.class）之间的关系很少需要显式地建模，尽管为了可视化系统的物理配置，有时这样做还是有用的。另一方面，通过用制品图对网页和其他可执行的制品进行建模，来可视化这种基于Web的系统的组织则比较常见。

---

[1]. 作者在本版中将第1版中所称的“进程视图”修改为“交互视图”（见本书第2章），但是在后面个别章节的叙述中没有彻底按照新的提法进行修改。这属于技术上的疏漏，译文进行了订正。——译者注

[2]. 原文如此。其实UML的依赖关系总是有方向的。——译者注

[3]. 将sequence diagram译为顺序图，是本书极个别与国标不一致之处。在制定国标GB/T 11457——2006时，对这种图的名称有“顺序图”和“时序图”两种不同意见，最后定为“时序图”。但是后来UML2.0又定义了另外一种图，即timing diagram，而国内的其他文献又把这种新的图译为“时序图”。于是这两种图的中文译名发生了冲突。为了避免由此引起的术语混乱，本书将它们分别译为“顺序图”和“定时图”。——译者注

第二部分 对基本结构建模

第4章 类

本章内容

类、属性、操作和责任

对系统的词汇建模

对系统中职责的分布建模

对非软件事物建模

对简单类型建模

进行高质量的抽象

类是任何面向对象系统中最重要的构造块。类是对一组具有相同属性、操作、关系和语义的对象的描述。一个类可以实现一个或多个接口。

类可以用来捕获正在开发的系统中的词汇。这些类可以包括作为问题域一部分的抽象，也可以包括构成实现的那些类。可以用类描述软件事物和硬件事物，甚至也可以用类描述纯粹概念性的事物。

【第9章讨论类的高级特性。】

结构良好的类具有清晰的边界，并形成了整个系统的职责均衡分布的一部分。

4.1 入门

对系统建模需要识别出关于特定视图的重要事物，这些事物形成了正在建模的系统的词汇。假如正在建造一所房子，那么像墙、门、窗户、柜橱和灯对于房主来说就是重要的事物。这些事物中的每一个都有别于其他事物，有自己的一组特性。墙有高度和宽度，是立体的。门也有高度和宽度，是立体的，但它还有额外的行为，能朝一个方向打开。窗户与门类似，二者都是在墙上开的洞，但它们稍有不同。窗户通常（但不总是）设计得让人能向外看，而不是让人穿行。

墙、门和窗户很少作为个体单独存在，因此还必须要考虑怎样把这些事物的具体实例组合在一起。识别事物和选择它们之间所要建立的关系将受下述因素影响：希望如何使用住宅的各个房间，希望各房间如何连通，以及希望这种布局所形成的总体风格与感觉。

用户所关心的事物各不相同。例如，帮助建房的水管工对排水管、存水弯和通风口之类的事物感兴趣。作为房主，就不需要关心这些事物，只需关心水管工对这些东西的施工要满足要求，例如，排水管要安装在地板下，通风口要穿过屋顶。

在UML中，所有的这些事物都被建模为类。类是对词汇表中一些事物的抽象。类不是个体对象，而是描述一些对象的一个完整集合。这样，可以在概念上把墙看成一个对象类，它具有一定的共同属性，如高度、长度、厚度和能否承重等。也可以考虑墙的个体实例，如“书房西南角的墙”。

【第13章讨论对象。】

在软件中，有很多编程语言直接支持类的概念。这很好，因为这意味着所建立的抽象经常可以直接地映射到编程语言，甚至可用于对非软件事物的抽象，如“顾客”、“交易”和“会话”等。

UML为类提供了图形表示，如图4-1所示。通过这种表示法能够独立于任何编程语言来对抽象进行可视化，并强调抽象的最重要的部分：名称、属性和操作。

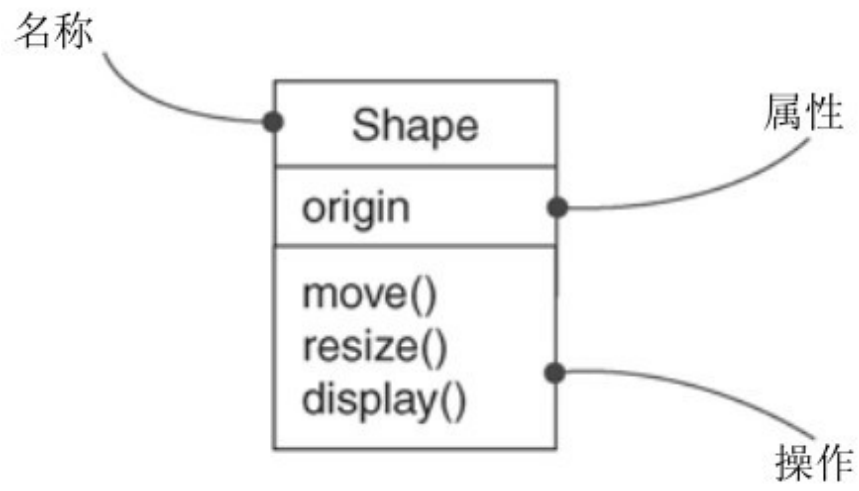


图4-1 类

4.2 术语和概念

类 （class）是对一组具有相同属性、操作、关系和语义的对象的描述。在图形上，把类画成一个矩形。

4.2.1 名称

每个类都必须有一个有别于其他类的名称。名称 （name）是一个文字串。单独的名称叫做简单名 （simple name），用类所在的包的名称作为前缀的类名叫做限定名 （qualified name）。绘制的类可以仅显示它的名称，如图4-2所示。

【一个包中的各类的名称都必须是唯一的，第12章要对此进行讨论。】

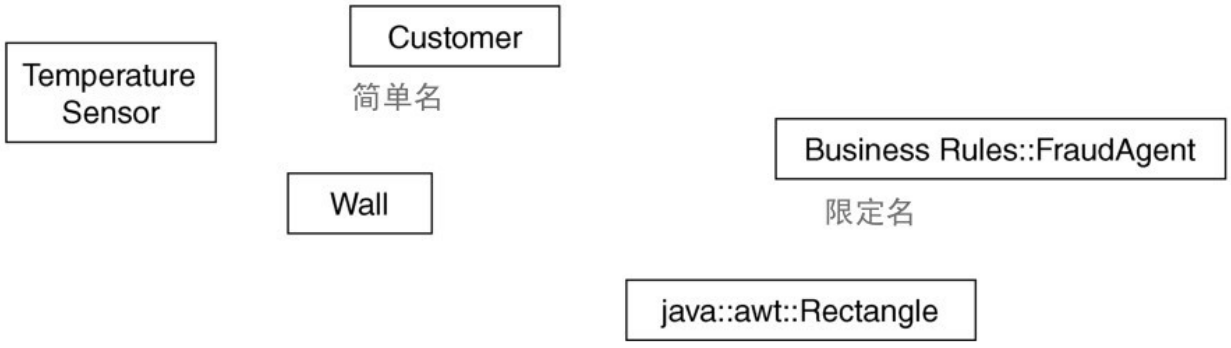


图4-2 简单名和限定名

注解 类名可以是由任何数目的字母、数字和某些标点符号（有些符号除外，例如用于分隔类名和包名的双冒号）组成的文本，它可以延伸成几行。在实践中，类名是从正在建模的系统的词汇表中提取出来的简单名词或名词短语。类名中的每个词的第一个字母通常要大写，如Customer或TemperatureSensor。

4.2.2 属性

属性（attribute）是已命名的类的特性，它描述了该特性的实例可以取值的范围。类可以有任意数目的属性，也可以没有属性。属性描述了正被建模的事物的一些特性，这些特性为类的所有对象所共有。例如，每一面墙都有高度、宽度和厚度；也可以用这样的方式对顾客建模：每个顾客都有一个姓名、地址、电话号码和出生日期。因此，属性是对类的对象可能包含的数据种类或状态种类的抽象。在一个给定的时刻，类的一个对象将具有该类的每一个属性的特定值。在图形上，将属性在类名下面的栏中列出。可以仅显示属性的名称，如图4-3所示。

【属性与聚合的语义有关，在第10章对此进行讨论。】

注解 属性名可以是像类名那样的文字。在实际应用中，属性名是描述属性所在类的一些特性的简短名词或名词短语。通常要将属性名中除第一个词之外的每个词的第一个字母大写，例如name或loadBearing。

可以通过声明属性的类以及属性可能的默认初始值来进一步地详述属性，如图4-4所示。

【可以详述属性的其他特征，例如可把它标记成只读的，或者是由本类的所有对象共享的，这些在第9章讨论。】

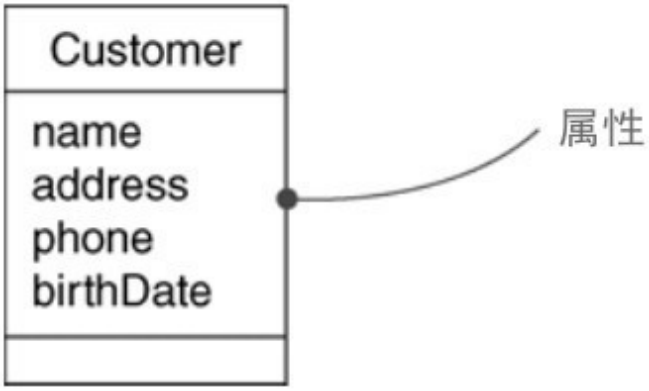


图4-3 属性

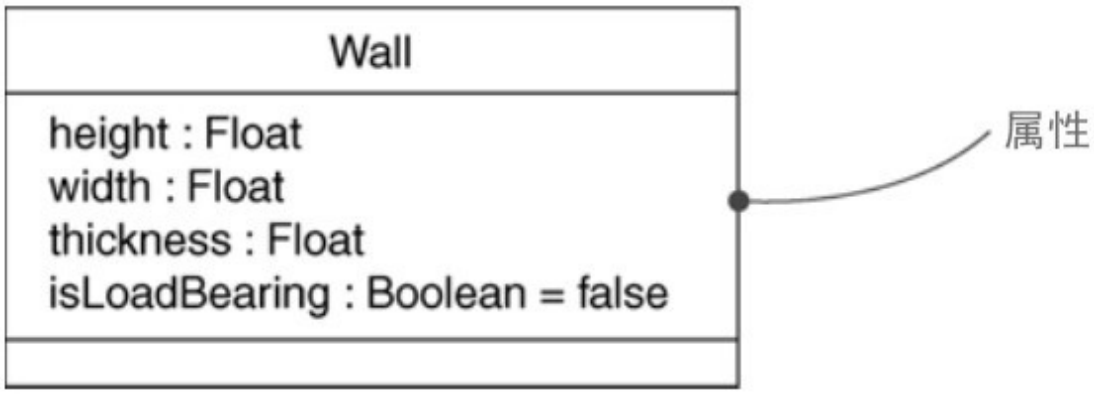


图4-4 属性和它们的类

4.2.3 操作

操作（operation）是一个服务的实现，该服务可以由任何类的对象来请求以影响其行为。换句话说，操作是能对一个对象所做的事情的抽象，并且它由这个类的所有对象共享。类可以有任意数目的操作，也可以没有操作。例如，在Java的awt包中的窗口库里，类Rectangle的所有对象都能被移动或调整大小，还可以查询它们的特性。调用对象的操作经常（但不总是）会改变该对象的数据或状态。在图形上，把操作列在类的属性栏下面的栏中。可以仅显示操作的名称，如图4-5所示。

【可以进一步用注释或活动图详述操作的实现，在第6章描述注释，在第20章讨论活动图。】

注解 操作名可以是像类名那样的文字。在实践中，操作名是描述它所在类的一些行为的短动词或动词短语。通常要将操作名中除第一个词之外的每个词的第一个字母大写，如move或isEmpty。

可以通过阐明操作的特征标记来详述操作，特征标记包含所有参数的名称、类型和默认值，如果是函数，还要包括返回类型，如图4-6所示。

【可以详述操作的其他特性，例如把操作标记为多态的、不变的或描述它的可见性，这些在第9章讨论。】

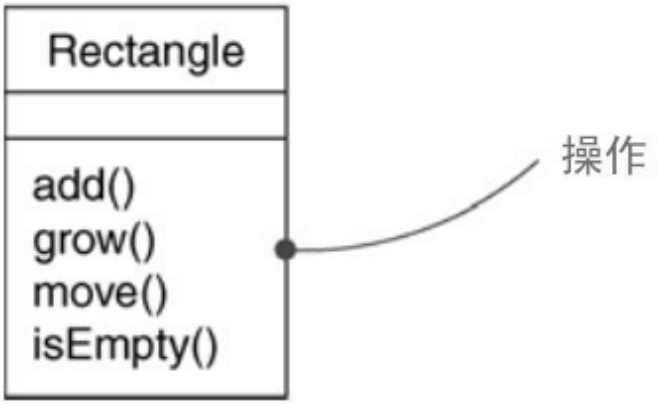


图4-5 操作

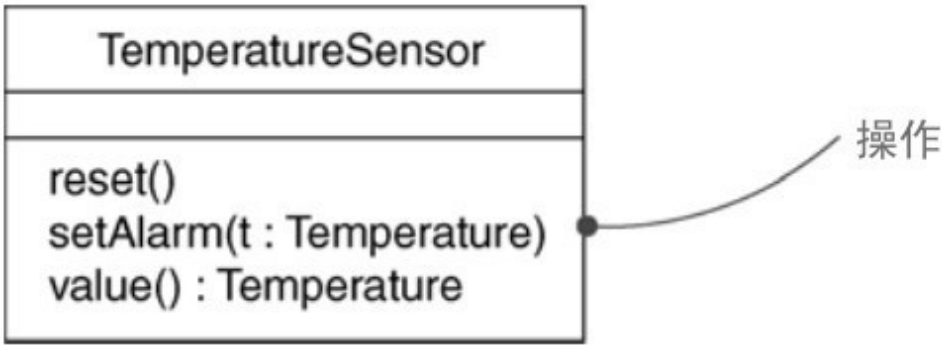


图4-6 操作和它们的特征标记

4.2.4 对属性和操作的组织

当画一个类时，不必同时把所有的属性和操作都显示出来。事实上，在大多数情况下不能这样做（由于属性和操作太多以致无法把它

们放在一张图中），也可能不应该这样做（可能只有这些属性和操作的一个子集与特定的视图相关）。由于这些原因，可以对类进行省略，这意味着可以有选择地仅显示类的一部分属性和操作，甚至可以一个也不显示。空栏并不一定意味着没有属性或操作，只是没有选择要显示它们。通过在列表的末尾使用省略号（“...”），可以明确地表示出实际的属性和操作比所显示的要多。也可以完全压缩掉这一栏，此时就不能表达是否有或者有多少属性或操作。

为了更好地组织属性和操作的长列表，可以利用衍型在每一组属性和操作之前加一个描述其种类的前缀，如图4-7所示。

【第6章讨论衍型。】

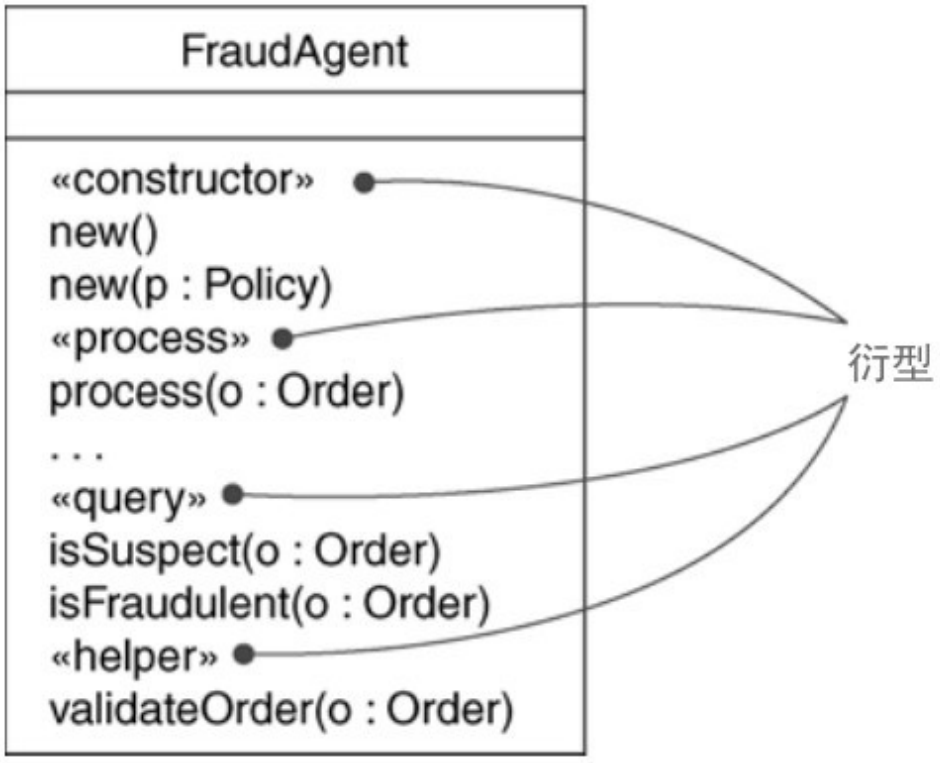


图4-7 用于类特征的衍型

4.2.5 职责

职责 （responsibility）是类的合约或责任。当创建一个类时，就声明了这个类的所有对象具有相同种类的状态和相同种类的行为。在较

高的抽象层次上，这些相应的属性和操作正是要完成类的职责的特征。类Wall负责了解墙的高度、宽度和厚度；在信用卡应用系统中，类FraudAgent负责处理汇票，并决定汇票是否合法、是否值得怀疑或是否具有欺诈性；类TemperatureSensor负责测量温度，若温度达到一定度数，就要发出警报。

【职责是一个已定义的衍型的例子，在第6章讨论。】

对类建模的一个好的起点是详述词汇表中的事物的职责。像CRC卡和基于用况的分析等技术在这里特别有用。类可以有任何数目的职责，尽管在实用中每个结构良好的类都最少有一个职责，而且最多也是可数的。当精化模型时，要把这些职责转换成能很好地完成这些职责的一组属性和操作。

【第9章讨论对类的语义建模。】

在图形上，把职责列在类图符底部的单独的栏中，如图4-8所示。

【也可以在注解中描绘出类的职责，在第6章讨论。】

图4-8 职责

注解 职责是自由形式的文本。实际上，可以把单个的职责写成一个短语、一个句子或（最多）一段短文。

4.2.6 其他特征

属性、操作和职责是创建抽象所需要的最常见的特征。事实上，对于大多数要建造的模型，这3种特征的基本形式足以表达类的最重要的语义。然而，有时需要可视化或详述其他特征，例如，单个属性和操作的可见性；与特定语言相关的操作特征，例如多态的或静态的；甚至类的对象可能产生或操纵的异常事件。在UML中能够表达这些以及很多其他特征，但它们被作为高级概念处理。

【第9章讨论高级的类概念。】

在建造模型时，很快会发现，创建的几乎每个抽象都是某种类型的类。有时要把类的实现与规约相分离，对此，在UML中可以用接口来表示。

【第11章讨论接口。】

在开始设计类的实现时，需要将其内部结构建模为一组连接起来的部件。为了得到最终的设计，可以把顶层类扩展成几层内部结构。

【第15章讨论内部结构。】

当开始建立更为复杂的模型时，可能会一次又一次地遇见同样的实体，如描述并发进程和线程的类或描述物理事物（例如 applet、Java Beans、文件、Web 页和硬件）的类目。因为这几种实体是很常见的，并且它们描述了重要的体系结构抽象，所以UML提供了主动类（表示进程和线程）和类目，例如制品（表示物理软件构件）和结点（表示硬件设备）。

【第23章、第25章和第27章讨论主动类、构件和结点，在第26章讨论制品。】

最后要说明的是，类很少单独存在。确切地讲，当建造模型时，通常要注重于相互作用的那些类群。在UML中，这些类的群体形成了协作，并且通常在类图中被可视化。

【第8章讨论类图。】

4.3 常用建模技术

4.3.1 对系统的词汇建模

类的最常见的用途是对从试图解决的问题或者从解决该问题的技术得到的抽象进行建模。每个这样的抽象都是系统词汇表的一部分，这意味着它们在整体上描述了对用户和实现者重要的事物。

对用户而言，大多数抽象并不难识别，因为这些抽象通常是从用户已经用来描述其系统的事物中抽取出来的。诸如CRC卡和基于用况的分析技术是帮助用户找到这些抽象的杰出方法。对于实现者来说，这些抽象通常是作为解决方案一部分的技术中的事物。

【第17章讨论用况。】

为了对系统的词汇建模，需做如下工作。

识别用户或实现者用于描述问题或者描述解决方案的那些事物。用 CRC 卡和基于用况分析的技术帮助用户发现这些抽象。

对于每个抽象，识别一个职责集。确保能清楚地定义每个类，而且这些职责能在所有的类之间很好地均衡。

提供为实现每个类的职责所需的属性和操作。

图4-9描述了从一个零售系统抽取的一组类，其中包括Customer、Order和Product。这个图也包含了一些来自问题的词汇表中的其他的相关抽象，如 Shipment（用于跟踪订单）、Invoice（用于按订单开发票）和Warehouse（在发货之前储存货物的地方）。还有一个与解决方案相关的抽象Transaction，用于订货和发货。

随着模型的不断增大，所发现的很多类将趋于簇集到一些在概念和语义上相关的组中。在UML中，可以用包来对这些类簇建模。

【第12章讨论包。】

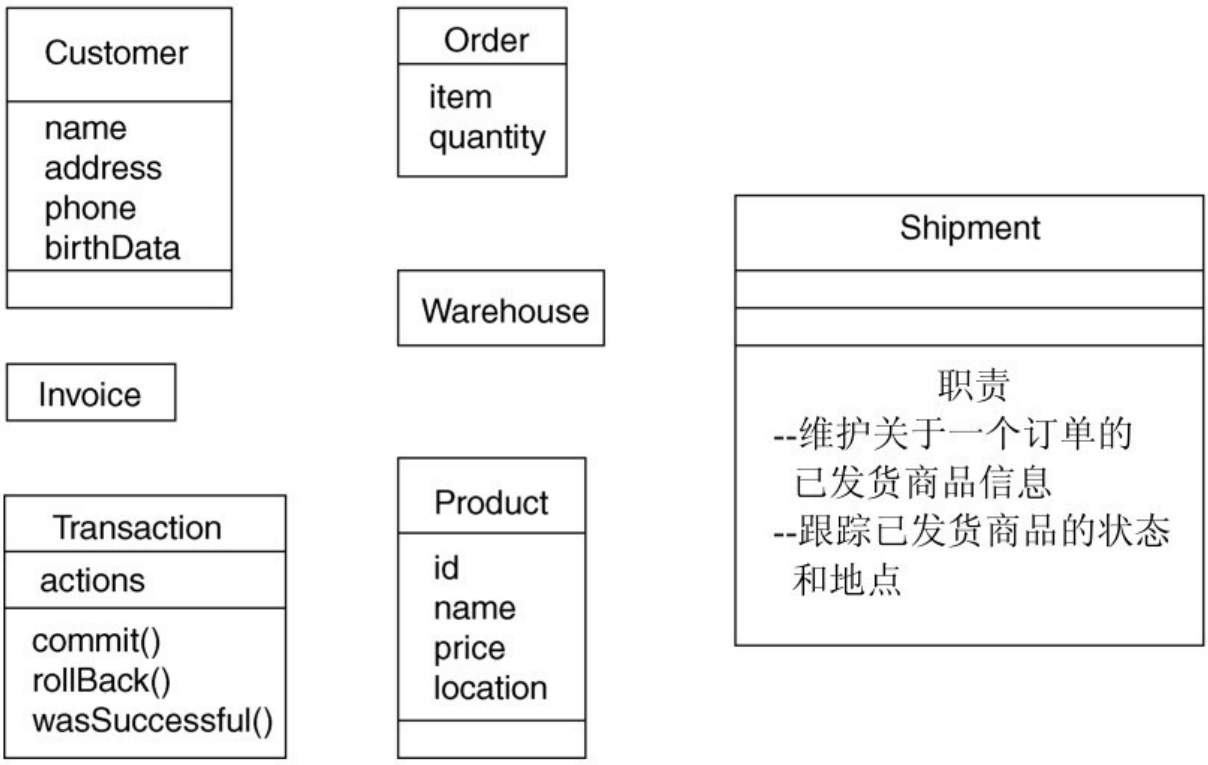


图4-9 对系统的词汇建模

完全静态的模型是很少的，相反，系统词汇中的大多数抽象都动态地相互作用。在UML中，有一些对这种动态行为建模的办法。

【本书的第四部分和第五部分讨论对行为建模。】

4.3.2 对系统中的职责分布建模

一旦开始对大量的类建模，就要保证抽象提供了均衡的职责集。这意味着不能让任何类过大或过小，每一个类应该做好一件事。若抽象出来的类过大，将会发现模型难以变化而且很不容易复用；若抽象出来的类过小，则最终抽象会过多，难以合理地管理和理解。可以使用UML来帮助可视化和详述这种职责的均衡。

对系统中的职责分布建模，要做如下工作。

识别一组为了完成某些行为而紧密地协同工作的类。

对上述的每一个类识别出一组职责。

从整体上观察这组类，把职责过多的类分解成较小的抽象，把职责过于琐碎的小类合成较大的类，重新分配职责以使每一个抽象合理地存在。

考虑这些类的相互协作方式，相应地重新分配它们的职责，使协作中没有哪个类的职责过多或过少。

【第28章讨论协作。】

例如，图4-10 展示了一组取自 Smalltalk 的类，图中显示了在类 Model、类 View 和类Controller中的职责分布。请注意所有的这些类如何在一起工作，使得其中没有过大或过小的类。

【这组类形成一个模式，这将在第29章进行讨论。】

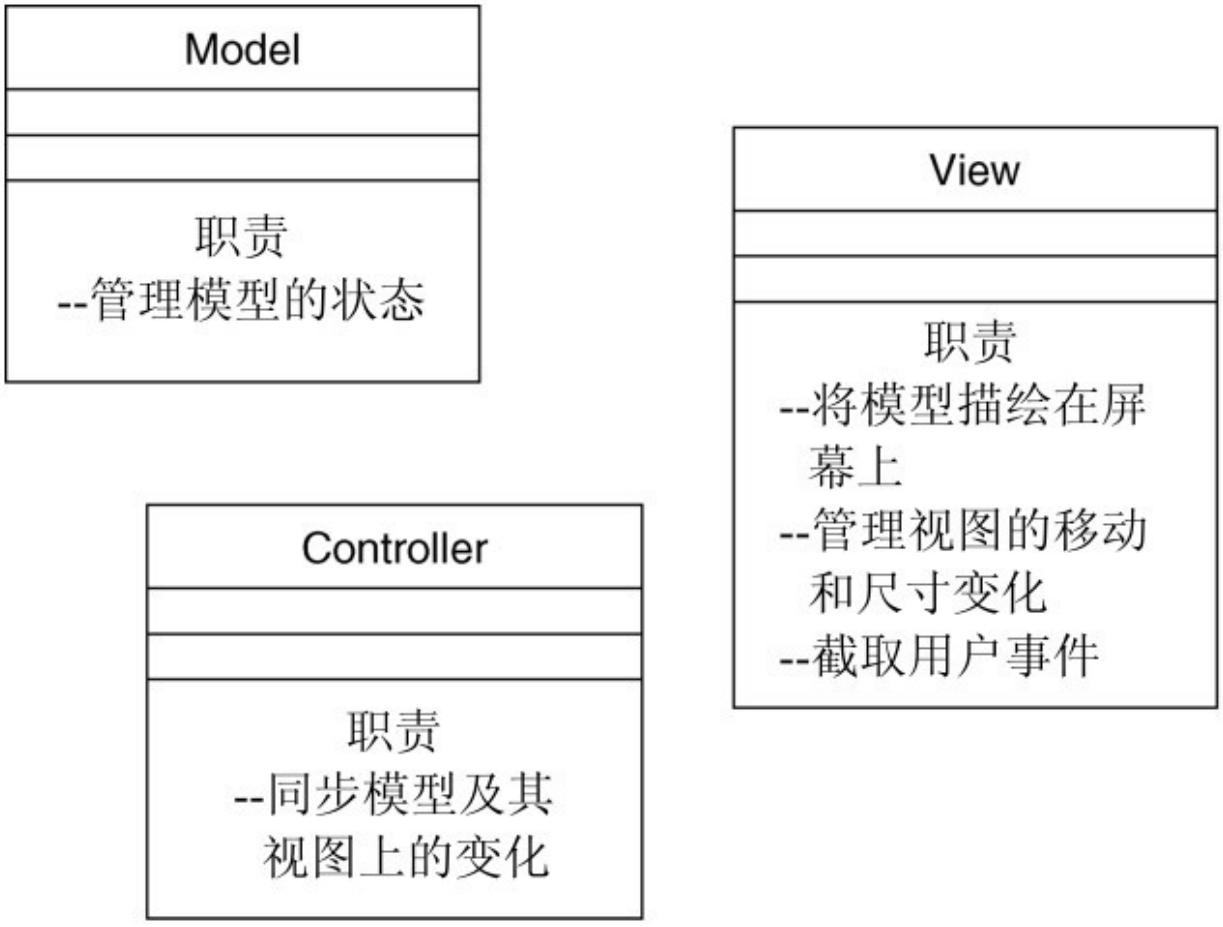


图4-10 对系统中的职责分布建模

4.3.3 对非软件事物建模

有时，要建模的事物在软件中并无类似物。例如，送发票的人和在仓库中按订单对货物进行自动包装以供发货的机器人可能是所建模的零售系统中的工作流的一部分。应用系统可能没有任何描述它们的软件（与上述例子中的顾客不同，因为系统可能要维护关于顾客的信息）。

为了对非软件事物建模，要做如下工作。

对抽象为类的事物建模。

如果要将这些非软件事物与UML已定义的构造块相区别，就要创建一个新的构造块，用衍型详述这些新语义，并给出不同的可视化提示。

【第6章讨论衍型。】

如果建模的事物是某种本身包含软件的硬件，考虑把它建模为一种结点，以便能进一步扩充它的结构。

【第27章讨论结点。】

注解 UML 主要用于对软件密集型系统建模，但如果与文本型硬件建模语言（如VHDL）相结合，UML 对硬件系统建模也很有表达力。OMG 也研制了 UML 的扩展产品SysML，意在用于系统建模。

如图4-11所示，把人（如AccountsReceivableAgent）和硬件（如Robot）抽象成类是

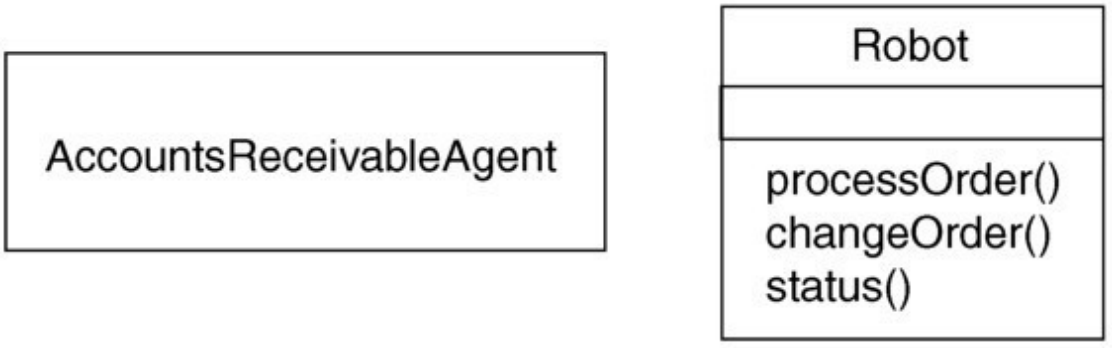


图4-11 对非软件事物建模

完全正常的，因为它们分别描述了具有共同结构和共同行为的一组对象。

【系统外部的事物经常被建模为参与者，这要在第17章进行讨论。】

4.3.4 对简单类型建模

在其他极端情况下，所建模的事物可能直接取自用于实现一个解的编程语言。通常这些抽象包括简单类型，如整数、字符和串，乃至自定义的枚举类型。

【第11章讨论类型。】

对简单类型建模，要做如下工作。

对抽象为类型或枚举的事物建模，这可以用带有适当衍型的类表示符来表示。

若需要详述与该类型相联系的值域，可以使用约束。

【第6章讨论约束。】

如图4-12所示，在UML中可以把这些事物建模为类型或枚举，就像类那样表示，但要显式地用衍型来做标记。把像整数（用类 Int 来表示）这样的简单类型建模为类型，可以用约束显式地说明这些事物的值域；必须在UML之外定义简单类型的语义。像Boolean和Status这样的枚举类型可以建模为枚举，并把它们的字面值罗列在属性分栏中（注意它们不是属性）。枚举类型也可定义操作。

【第11章讨论类型。】

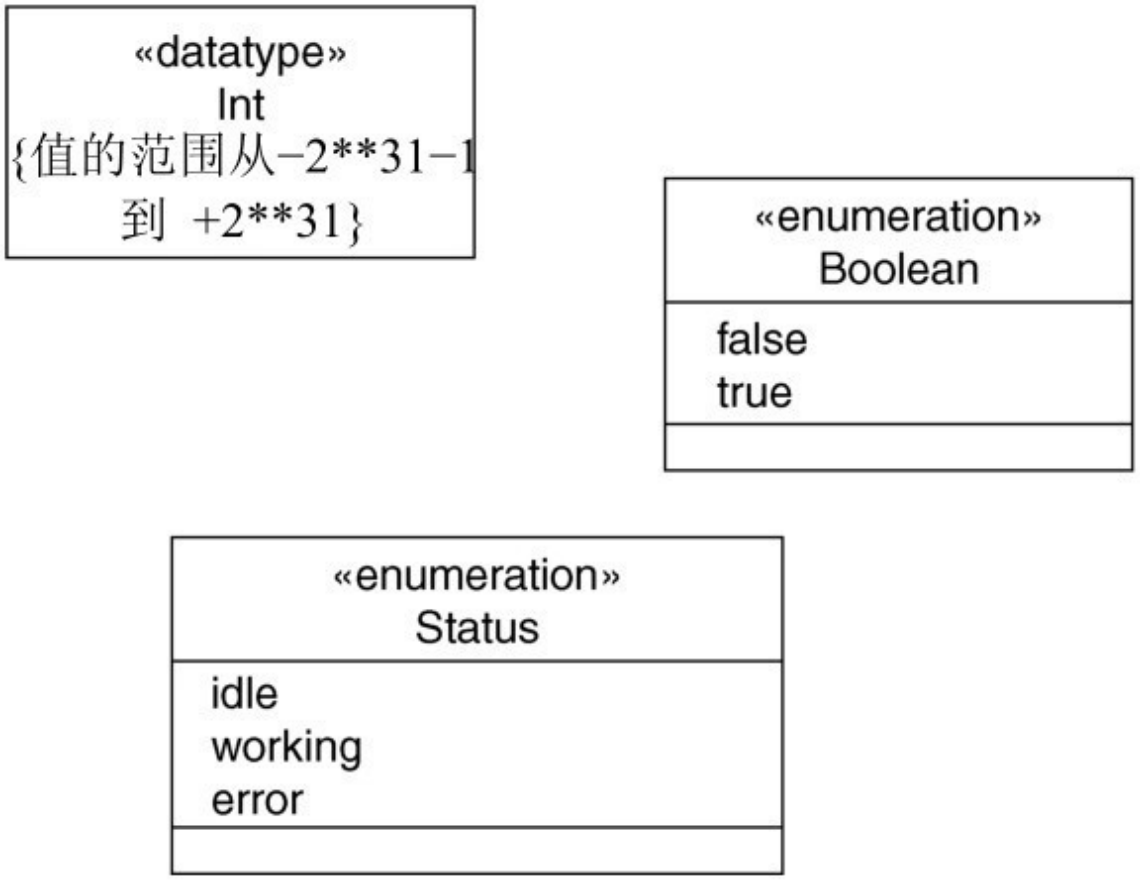


图4-12 对简单类型建模

注解 像C和C++这样的一些语言，对每一个枚举的字面值都设定一个整数值。在UML中，可通过对枚举的字面值附加注释以作为实现指导，来对此建模。整型值无需逻辑建模。

4.4 提示和技巧

在用UML对类建模时要记住：对最终用户或实现者来说，各个类都应该映射到某个有形的或者概念性的抽象。一个结构良好的类，应满足如下条件。

为取自问题域或者解域的词汇中的事物提供明确的抽象。

嵌入一个小的、明确定义的职责集，并且能很好地实现它们。

把抽象的规约和它的实现清楚地分开。

简单而且可理解，并具有可适应性和可扩展性。

当用UML绘制一个类时，要遵循如下策略。

仅显示在该类的语境中对于理解抽象较为重要的类的特性。

按属性和操作的种类进行分组，以更好地组织其长列表。

把相关的类显示在同一个类图中。

第5章 关系

本章内容

依赖、泛化和关联关系

对简单依赖建模

对单继承建模

对结构关系建模

创建关系网

当建造抽象时，会发现类很少单独存在，相反，大多数类以几种方式相互协作。因此，当对系统建模时，不仅要识别形成系统词汇的事物，而且还必须对这些事物如何相互联系建模。

在面向对象的建模中，有3种特别重要的关系：依赖（dependency），它表示类之间的使用关系（包括精化、跟踪和绑定关系）；泛化（generalization），它把一般类连接到它的特殊类；关联（association），它表示对象之间的结构关系。其中的每一种关系都为组合抽象提供了不同的方法。

【第10章讨论关系的高级特征。】

构造关系网与创建类之间职责的均衡分布一样。过分细致的设计，将导致关系混乱，使得模型不可理解；对设计考虑得过少，将会丢失系统中事物协作方式所蕴涵的许多有用信息。

5.1 入门

如果正在建造一所房子，像墙、门、窗户、橱柜和照明灯具这样的事物将形成部分词汇。然而这些事物都不是单独存在的。墙要与别的墙相连接；门和窗要安在墙上，分别形成供人们出入和采光的开口；橱柜和照明灯具自然要安在墙上和天花板上。把墙、门、窗户、橱柜和照明灯具组合在一起，就形成了像房间这样较高层次的事物。

在这些事物中，不仅能发现结构关系，而且也能发现其他种类的关系。例如，房子肯定有窗户，但窗户的种类可能有很多。可能有不能开的凸型大窗户和能开的小厨房窗户；一些窗户能上下开，而另一些窗户（像通向庭院的窗户）可以左右拉开；一些窗户仅有一块玻璃，另一些窗户有两块玻璃。无论它们多么不同，它们都具有一些基本的窗户要素：每个窗户都是墙上的一个开口，用来采光和通气，有时还能过人。

在UML中，事物之间这些相互联系的方式（无论是逻辑上的还是物理上的）都被建模为关系。在面向对象的建模中，有3种最重要的关系：依赖、关联和泛化。

（1）依赖 （dependency）是使用关系。例如，水管依赖热水器，对它们所运送的水进行加热。

（2）关联 （association）是实例之间的结构关系。例如，房间是由墙和一些其他事物组成的，墙上可以镶嵌门和窗，管道可以穿过墙体。

（3）泛化 （generalization）把一般类连接到较为特殊的类，也称为超类/子类关系或父/子关系。例如，观景窗是一种带有固定的大窗格的窗户，庭院窗是一种带有向两边开的窗格的窗户。

这3种关系覆盖了大部分事物之间相互协作的重要方式。显然，这3种关系也能很好地映射到大多数面向对象编程语言所提供的连接对象

的方式。

【另外几种关系（如实现和精化）在第10章讨论。】

UML 对每种关系都提供了一种图形表示，如图5-1 所示。这种表示法允许脱离具体的编程语言而对关系进行可视化，可用以强调关系的最重要的部分：关系名、关系所连接的事物和关系的特性。

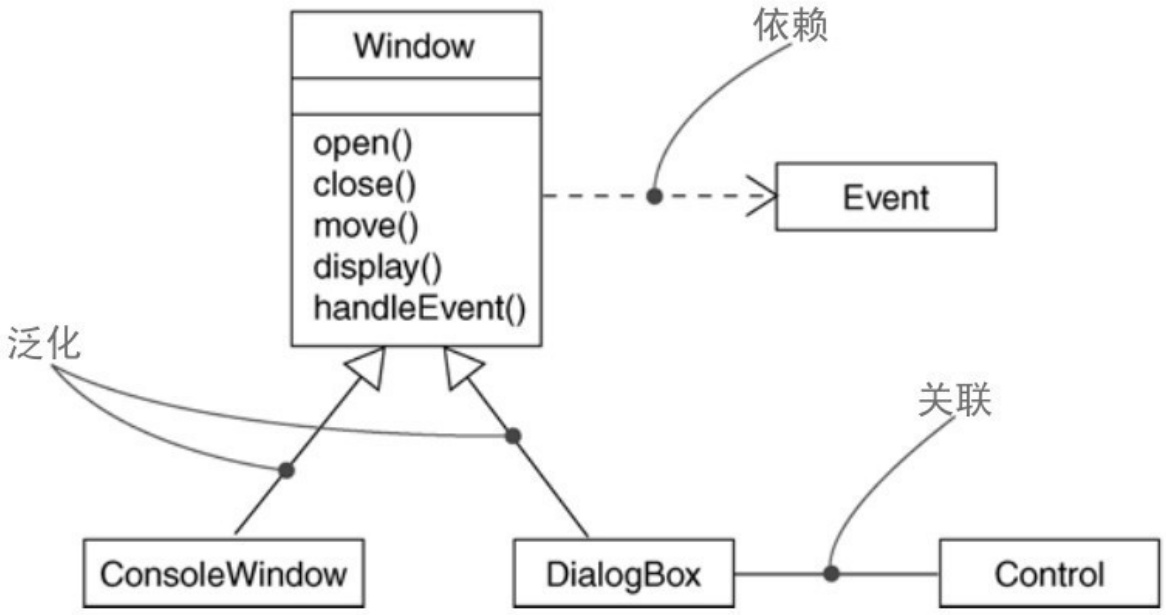


图5-1 关系

5.2 术语和概念

关系 （relationship）是事物之间的联系。在面向对象的建模中，最重要的3种关系是依赖、泛化和关联。在图形上，把关系画成一条线，并用不同的线区别关系的种类。

5.2.1 依赖

依赖 （dependency）是一种使用关系，说明一个事物（如类Window）使用另一个事物（如类Event）的信息和服务，但反之未必。在图形上，把依赖画成一条有向的虚线，指向被依赖的事物。当要指明一个事物使用另一个事物时，就选用依赖。

在大多数情况下，在类与类之间用依赖指明一个类使用另一个类的操作，或者使用其他类所定义的变量和参量，参见图5-2。这的确是一种使用关系，如果被使用的类发生变化，那么另一个类的操作也会受到影响，因为这个被使用的类此时可能表现出不同的接口或行为。在UML中，也可以在很多其他的事物之间创建依赖，特别是注解和包。

【第6章讨论注解，第12章讨论包。】

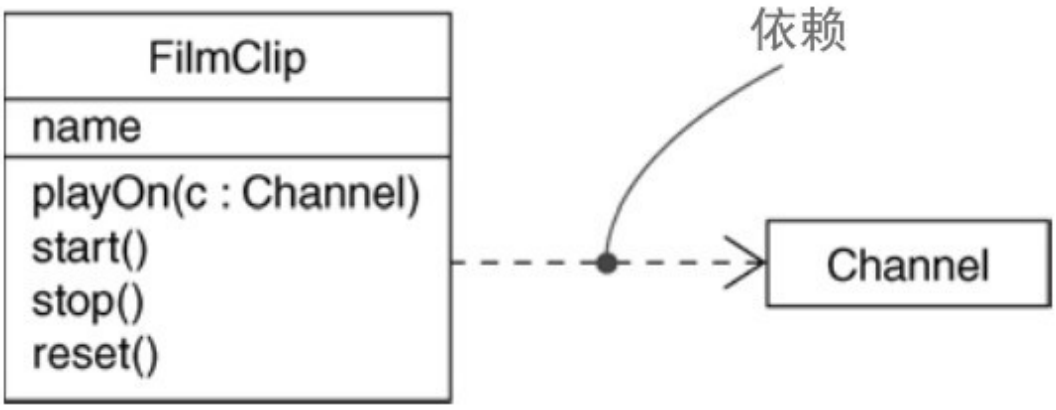


图5-2 依赖

注解 依赖可以带有一个名字，但很少使用，除非模型有很多依赖，并且要引用它们或做出区别。在一般情况下，用衍型区别依赖的不同含义。

【第10章讨论不同种类的依赖，第6章讨论衍型。】

5.2.2 泛化

泛化 （generalization）是一般事物（称为超类或父类）和该事物的较为特殊的种类（称为子类或子）之间的关系。有时也称泛化为“is-a-kind-of” [1] 关系：一个事物（如类BayWindow）是更一般的事物（如类 Window）的“一个种类”。泛化意味着子类的对象可以被用在父类的对象可能出现的任何地方，反之则不然。换句话说，泛化意味着子类可以替换父类的声明。子类继承父类的特性，特别是父类的属性和操

作。通常（但不总是），子类除了具有父类的属性和操作外，还具有更多的属性和操作。若子类的一个操作的实现覆盖了父类的同样一个操作的实现，则这种情况称为多态性。其共同之处是，两个操作必须具有相同的特征标记（相同的名字和参数）。在图形上，把泛化画成一条带有空心三角形大箭头的有向实线，指向父类，如图5-3所示。当要表示父/子关系时，就使用泛化。

一个类可以有0个、1个或多个父类。没有父类并且最少有一个子类的类称为根类或基类；没有子类的类称为叶子类。如果一个类只有一个父类，则说它使用了单继承；如果一个类有多个父类，则说它使用了多继承。

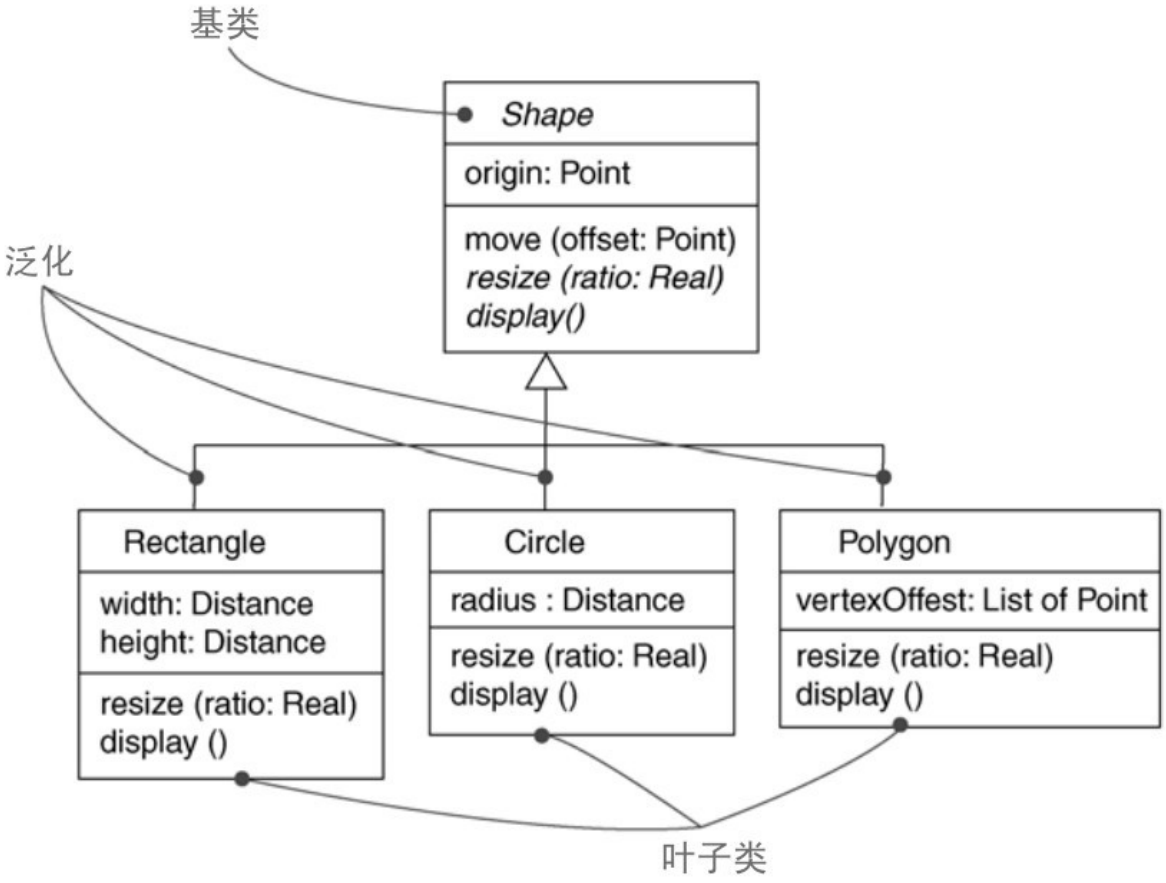


图5-3 泛化

在大多数情况下，用类或接口之间的泛化来表明继承关系。在UML中，也可以在其他的类目之间创建泛化，比如结点之间。

【第12章讨论包。】

注解 带有名字的泛化指明对一个父类的子类从特定方面的划分，称为泛化集。多个泛化集是正交的；用多继承从每个泛化集中选出一个子类来特别指明超类。以上属于高级主题，本书没有涵盖。

5.2.3 关联

关联 （association）是一种结构关系，它指明一个事物的对象与另一个事物的对象间的联系。给定一个连接两个类的关联，可以从一个类的对象联系到另一个类的对象。关联的两端都连到同一个类是完全合法的。这意味着，从类的一个给定对象能连接到该类的其他对象。恰好连接两个类的关联叫做二元关联。尽管不太常见，但可以有连接多于两个类的关联，这种关联叫做n元关联。在图形上，把关联画成一条连接相同类或不同类的实线。当要表示结构关系时，就使用关联。

【关联和依赖可以是反射的，但泛化关系不然，这在第10章讨论。】

除了这种基本形式外，还有4种应用于关联的修饰。

1.名称

关联可以有一个名称，用以描述该关系的性质。为了消除名称的歧义，可提供一个指出读名称方向的三角形，给名称一个方向，如图5-4所示。

【不要把名称方向和关联导航相混淆，在第10章讨论这方面的问题。】

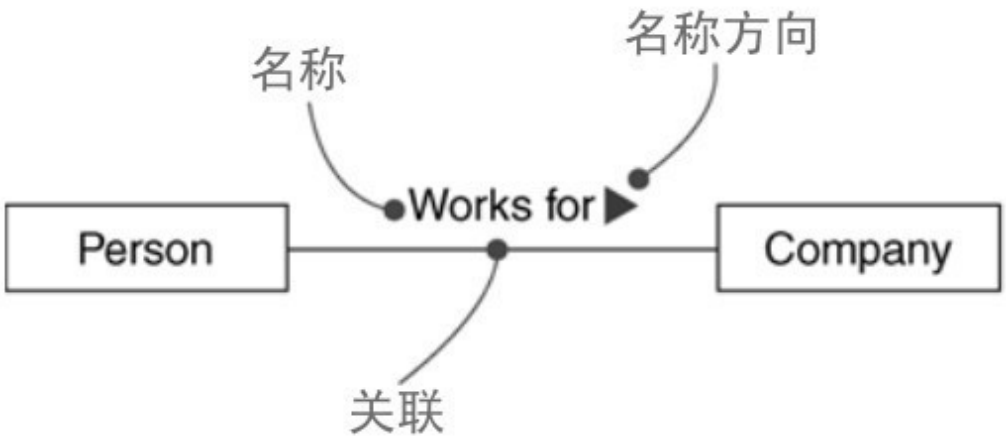


图5-4 关联的名称

注解 虽然关联可以有名称，但在显式地给出关联的端点名的情况下通常不需要给出关联名称。若用多个关联连接同一个类，有必要使用关联名或关联端点名来区分它们。若一个关联有多于一个端点是在同一个类上，有必要使用关联端点名来区分端点。若两个类之间只有一个关联，一些建模者省去了关联的名称，但为了使关联的用意清晰最好使用关联名。

2.角色

当一个类参与了一个关联时，它就在这个关系中扮演了一个特定的角色。角色是关联中靠近它的一端的类对另一端的类呈现的面孔。可以显式地命名一个类在关联中所扮演的角色。把关联端点扮演的角色称为端点名（在UML1中称为角色名）。在图5-5中，扮演 employee 角色的类Person与扮演employer角色的类Company相关联。

【角色与接口的语义相关，这在第11章讨论。】

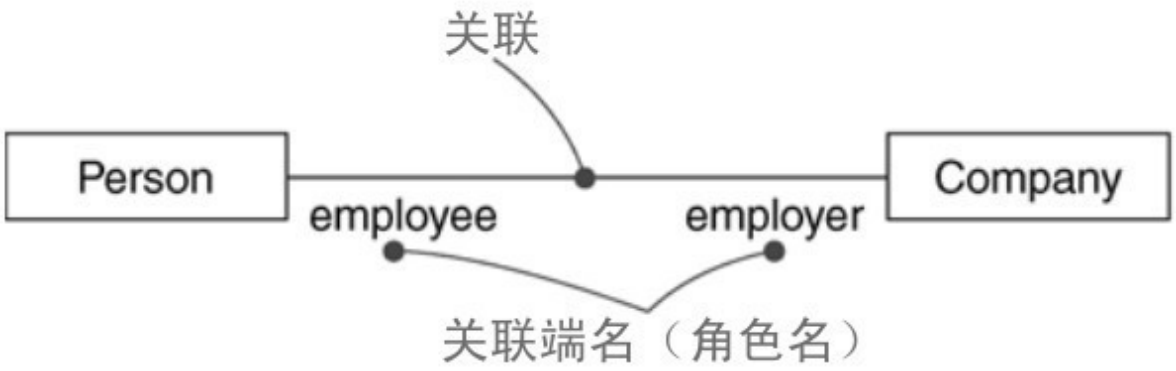


图5-5 关联端名（角色名）

注解 同一个类可以在其他关联中扮演相同或不同的角色。

注解 可以把属性看作类拥有的单向关联。该属性的名称与关联彼端的端点名相一致。

3.多重性

关联表示了对象间的结构关系。在很多建模问题中，说明一个关联的实例中有多少个相互连接的对象是很重要的。这个“多少”被称为关联角色的多重性，它表示一个整数的范围，指明一组相关对象的可能个数。将多重性写成一个表示取值范围的表达式，其最大值和最小值可以相同，用两个圆点把它们分开。声明了关联一端的多重性，这说明：对于关联另一端的类的每个对象，本端的类可能有多少个对象出现。对象数目必须是在给定的范围内。可以精确地表示多重性为：一个（1）、零个或一个（0..1）、多个（0..*）、一个或多个（1..*）。可以给出它的一个整数范围（如2..5），甚至可以精确地指定多重性为一个数值（如3与3..3等价）。

【关联的实例称为链，在第16章讨论。】

如图5-6所示，每个公司对象可以雇佣一个或多个人员对象（多重性为1..*）；每个人员对象受雇于0个或多个公司对象（多重性为*，它等价于0..*）。

4.聚合

两个类之间的简单关联表示了两个同等地位的类之间的结构关系，这意味着这两个类在概念上是同级别的，一个类并不比另一个类更重要。有时要对“整体/部分”关系建模，其中一个类描述了一个较大的事物（“整体”），它由较小的事物（“部分”）组成。这种关系称为聚合，它描述了“has-a”关系，意思是整体对象拥有部分对象。其实聚合只是一种特殊的关联，它被表示为在整体的一端用一个空心菱形修饰的简单关联，如图5-7所示。

【聚合有一些重要的变种，这在第10章讨论。】

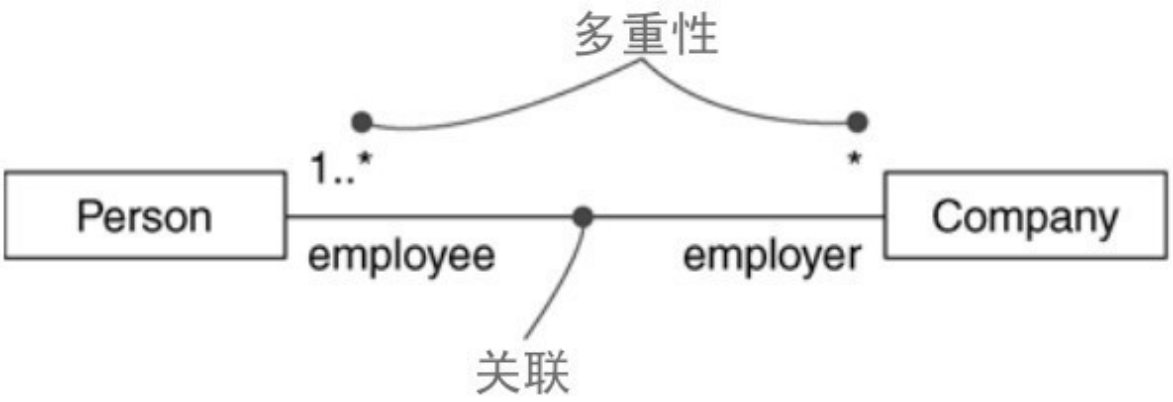


图5-6 多重性

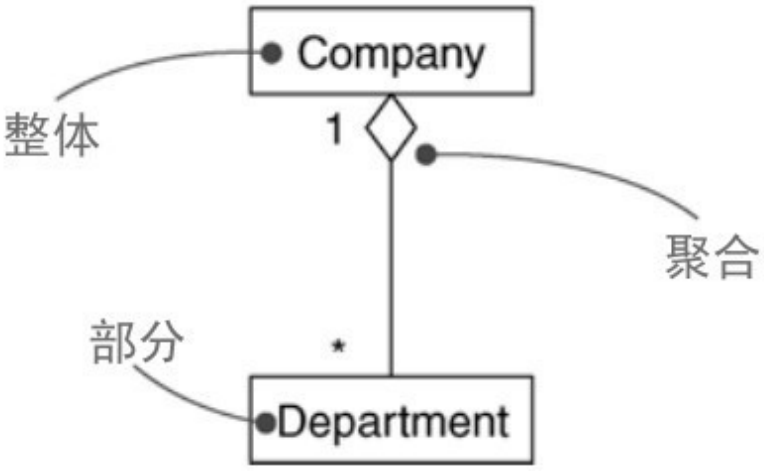


图5-7 聚合

注解 这种简单形式的聚合的含义完全是概念性的。空心菱形只是把整体和部分区别开来。这意味着简单聚合没有改变在整体与部分之

间整个关联的导航含义，也与整体和部分的生命期无关。关于更紧密的聚合形式，参看第10章关于组合的那一节。

5.2.4 其他特征

简单而未加修饰的依赖、泛化以及带有名称、多重性和角色的关联是创建抽象时所需要的最常见的特征。事实上，对于所建的大多数模型，这3种关系的基本形式足以表达关系的最重要的语义。然而，有时需要可视化或详述其他特征，如组合聚合、导航、判别式、关联类、特殊种类的依赖和泛化。这些以及很多其他的特征都可以用UML表达，但它们都被作为高级概念处理。

【第10章讨论高级关系概念。】

依赖、泛化和关联都是定义在类这一级别上的静态事物。在UML中，通常是在类图中对这些关系进行可视化。

【第8章讨论类图。】

当开始在对象级别上建模时，特别是开始解决这些对象的动态协作时，将遇到链（它是关联的实例，描述可能发送消息的对象间的连接）。

【第16章讨论链。】

5.2.5 绘图风格

用图符之间的连线来表示图中的关系。连线有不同的装饰（如箭头或菱形），以此来区分不同种类的关系。通常，建模者选用以下两种风格之一来绘制连线。

用任意角度的斜线。除非需要用多条线段来避开用其他图符，否则只用一条线段。

将直线画得与页边平行。除了用一条线段连接两个并排的图符的情况外，要将连线画成以直角连接的一组线段。这是本书中使用最多的一种风格。

只要小心，大多数的连线交叉是可以避免的。如果线交叉是必要的，为了避免相连的路径的不确定性，就用一个小弧来表示连线交叉，如图5-8所示。

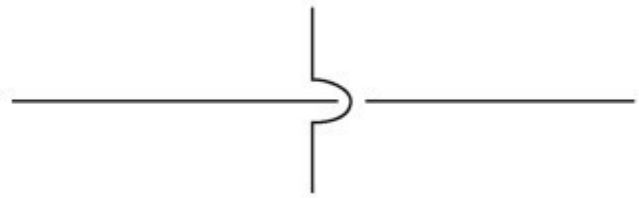


图5-8 连线交叉的符号

5.3 常用建模技术

5.3.1 对简单依赖建模

一种常见的依赖关系是两个类之间的连接，其中的一个类只是使用另一个类作为它的操作参数。

对这种使用关系建模，要做如下工作。

创建一个依赖，从含有操作的类指向被该操作用来作为参数的类。

例如，图5-9显示了取自一个大学管理学生选课和教师任课的系统中的一组类。图中显示了一个从CourseSchedule到Course的依赖，因为Course被用作CourseSchedule的操作add和remove的参数。

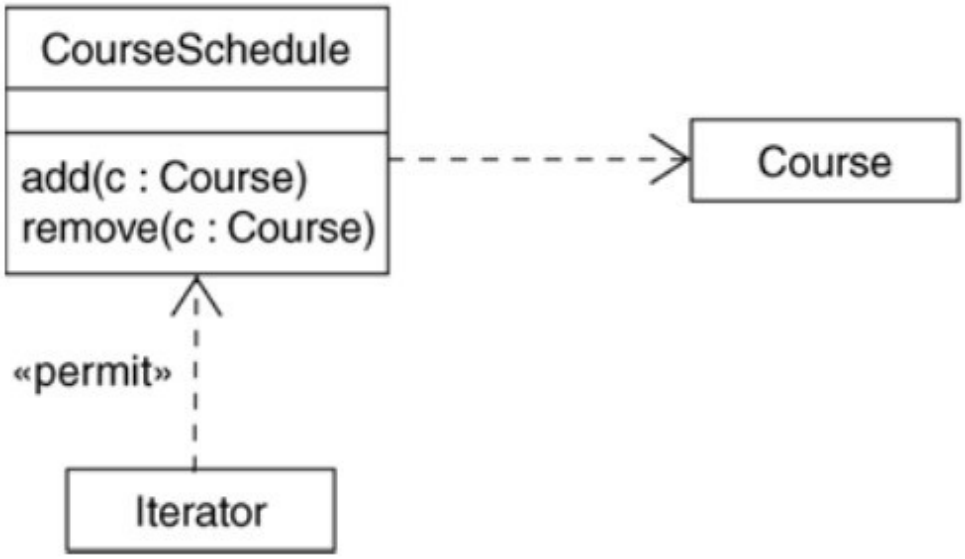


图5-9 依赖关系

如果像图5-9 这样给出了操作的完整的特征标记，一般就不需要给出这个依赖，这是因为对类的使用已经显式地写在特征标记中。然而有时要显示这样的依赖，特别是当省略了操作的特征标记，或模型描述了被使用类的其他关系时。

图5-9还显示了另一个依赖关系，这个依赖没有涉及操作中的类，而是对C++的一种习惯用法进行建模。这个起自Iterator的依赖表明Iterator使用CourseSchedule，但 CourseSchedule 不知道有关 Iterator 的任何信息。用一个衍型«permit»来标记这个依赖，它与C++中的friend声明类似。

【第10章讨论其他关系衍型。】

5.3.2 对单继承建模

在对系统的词汇建模中，经常会遇到在结构或行为上与其他的类相似的类。可以把这样的每一个类建模为独立的、不相关的抽象。但更好的方法是提取所有共同的结构特征和行为特征，并把它们提升到较为一般的类中，特殊类从中继承这些特征。

对继承关系建模，要做如下工作。

给定一组类，寻找两个或两个以上的类中的共同职责、属性和操作。

把这些共同的职责、属性和操作提升到较为一般的类中。如果需要，创建一个新类，用以指派这些元素（但要小心不要引入过多的层次）。

画出从每个特殊类到它的较一般的父类的泛化关系，用以表示较特殊的类继承较一般的类。

例如，图5-10显示了取自一个贸易应用中的一组类。可以看到从类CashAccount、Stock、Bond 和 Property 到较为一般的名为Security 的类的泛化关系。Security 是父类，CashAccount、Stock、Bond 和 Property 都是子类，每一个这样的特殊的子类都是一种Security。注意Security包括两个操作：presentValue和history。由于Security是这4个类的父类，因此CashAccount、Stock、Bond和Property都继承了这两个操作，同时也继承了可能在图5-10中省略了的Security的其他任何属性和操作。

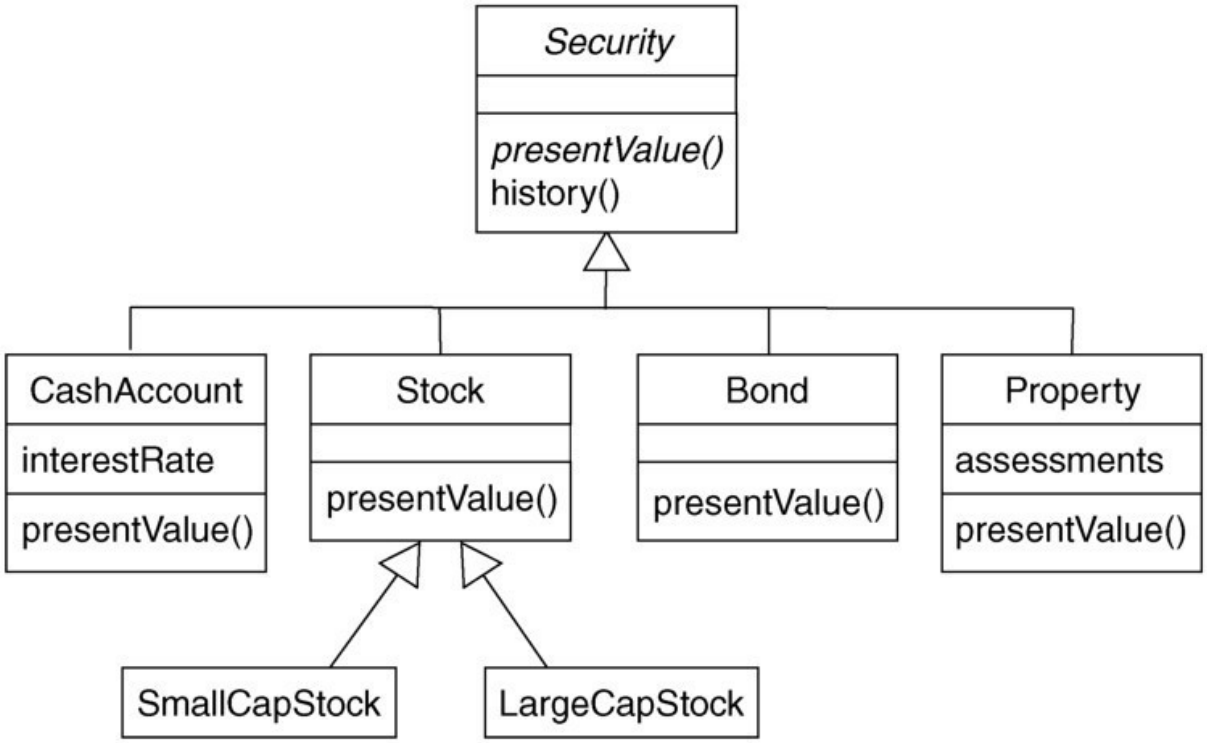


图5-10 继承关系

读者可能已注意到Security和presentValue的写法与其他类和操作的写法有所不同，这样做是有原因的。当建造像图5-10 那样的层次时，经常会遇到不完全的或不想让它有任何对象的非叶子类。通常把这样的类称为抽象类（abstract class）。在UML中，通过把类名写为斜体，以指明这个类是抽象的，例如类 Security 就是如此。这种规定也适用于操作，如presentValue，这意味着这样的操作提供了一种特征标记，但它是不完全的，因此必须在较低的抽象层次用一定的方法实现。事实上，如图5-10所示，Security的4个直接子类都是具体的（即非抽象的），并且也分别提供了操作presentValue的具体实现。

【第9章讨论抽象类和抽象操作。】

一般/特殊的层次不必仅限于两层。事实上，像图5-10中那样，多于两层的继承层次是很常见的。SmallCapStock和LargeCapStock是Stock的两个子类，Stock是Security的子类。由于Security没有父类，因此它是一个根类。由于SmallCapStock和LargeCapStock都没有子类，因此它们是叶子类。Stock既有父类也有子类，因此它既不是根类也不是叶子类。

尽管在本例中没有体现，但实际上也可以为一个类创建多个父类。这称为多继承，这意味着该类继承它的各个父类的所有属性、操作和关联。

【第10章讨论多继承。】

当然，在继承格中不能有任何循环，即一个给定的类不能是它自己的父类。

5.3.3 对结构关系建模

当用依赖或泛化关系建模时，可能是对表示了不同重要级别或不同抽象级别的类建模。给定两个类间的依赖，则一个类依赖另一个

类，但后者没有前者的任何信息。给定两个类间的泛化关系，则子类从它的父类继承，但父类没有任何子类所特有的信息。简而言之，依赖和泛化关系都是不对称的。

当用关联关系建模时，是在对相互同等的两个类建模。给定两个类间的关联，则这两个类以某种方式相互依赖，并且常常从两边都可以导航。依赖是使用关系，泛化是“is-a-kind-of”关系，而关联描述了类的对象之间相互作用的结构路径。

【关联在默认的情况下是双向的，也可以限制它们的方向，在第10章讨论这些问题。】

对结构关系建模，要做如下工作。

对于每一对类，如果需要从一个类的对象到另一个类的对象导航，就要在这两个类之间说明一个关联。这是关联的数据驱动观点。

对于每一对类，如果一个类的对象要与另一个类的对象相互交互，而后者不作为前者的过程局部变量或者操作参数，就要在这两个类间说明一个关联。这是关联的行为驱动观点。

对于这样的每一个关联，要说明其多重性（特别是当多重性不为*时，其中*是默认的多重性）和角色名（特别是在有助于解释模型的情况下）。

如果关联中的一个类与另一端的类相比，前者在结构或者组织上是一个整体，后者看起来像它的部分，则在靠近整体的一端用一个菱形对该关联进行修饰，从而把它标记为聚合。

怎样才能知道一个给定类的对象何时必须与另一个类的对象相互作用？答案是，CRC 卡和用况分析非常有助于考虑结构性和行为性脚本。在有两个或两个以上的类用数据关系进行交互的地方说明一个关联。

【第17章讨论用况。】

图5-11 所显示的是取自一个学校的信息系统中的一组类。从该图的左下部开始，可以找到名称为Student、Course和Instructor的类。在Student和Course之间有一个关联，它描述了学生参加的课程。同时，每一名学生可以参加任意门数的课程，而每一门课程可以由任意名学生参加。类似地，在Course和Instructor之间也有一个关联，它描述了教师所教的课程。每一门课至少有一名教师，而每一名教师可以教零到多门课。每门课精确地属于一个系。

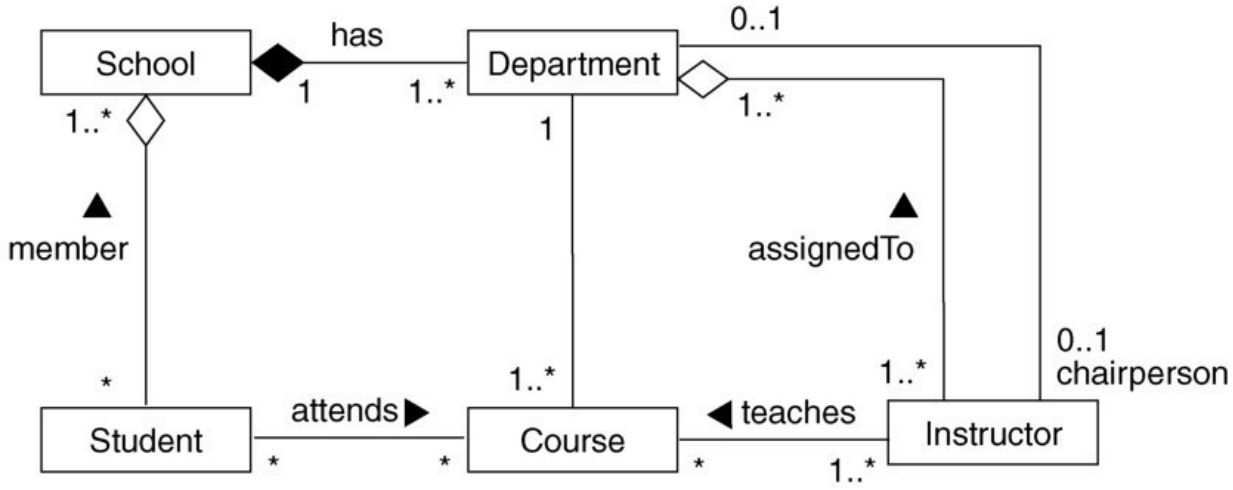


图5-11 结构关系

School和Student以及它和Department之间的关系有点不同。在这里可以看到聚合关系。一所学校可以有零到多名学生，一名学生可以是在一所或者多所学校注册的学员，一所学校可以有一个或多个系，每个系只能属于一所学校。可以不用聚合修饰而用简单的关联，但通过说明School是整体，Student和Department是部分，可以说清楚在组织上哪个高于哪个。因此，学校在一定程度上由学生和学生所在的系来定义。类似地，实际上学生和系并不是与他们所属的学校无关，而是从他们的学校能得到他们的身份。

【School和Department之间的聚合关系是组合聚合，在第10章讨论这个问题。组合是紧密形式的聚合，它包含一种拥有关系。】

还可以看到，在Department和Instructor之间有两个关联。其中的一个关联说明可以指派一名教师到一个或多个系中，而一个系可以有一名或多名教师。由于在学校的组织结构中系比教师的层次要高，所以这要用聚合来建模。另一个关联表明一个系只能有一名教师是系主任。这种建模方式说明，一名教师最多是一个系的系主任，并且某些教师不是任何系的系主任。

注解 学校可能没有系。系主任可能不是教师，甚至学生也可以是教师。这不意味着这个模型是错误的，只是学校的情况有所不同而已。不能孤立地建模，像这样的每一个模型都依赖于打算怎样使用这些模型。

5.4 提示和技巧

在用UML对关系建模时，要遵循如下策略。

仅当被建模的关系不是结构关系时，才使用依赖。

仅当关系是“is-a-kind-of”关系时，才使用泛化。往往可以用聚合代替多继承。

小心不要引入循环的泛化关系。

一般要保持泛化关系的平衡；继承的层次不要太深（大约多于 5 层就应该想一想），也不要太宽（代之以寻找可能的中间抽象类）。

关联主要用于对象间有结构关系的地方。不要用关联来表示暂时关系，例如过程的参数或局部变量。

在用UML绘制关系时，要遵循如下策略。

要一致地使用平直的线或斜线。平直的线给出的可视化提示强调了相关事物之间的连接都集中到一个共同事物。在复杂的图中斜线则经常有更好的空间效果。在同一个图中使用两种线型，有助于把人们的注意力引导到不同的关系组上。

除非绝对必要，否则要避免连线交叉。

仅显示对理解特定的成组事物必不可少的关系。避免使用多余的关系（特别是多余的关联）。

第6章 公共机制